

TWO DYNAMIC MODELS AND THEIR APPLICATION

Ferenc Belik



LUND UNIVERSITY

Department of Computer Sciences

1986

TWO DYNAMIC MODELS AND THEIR APPLICATION

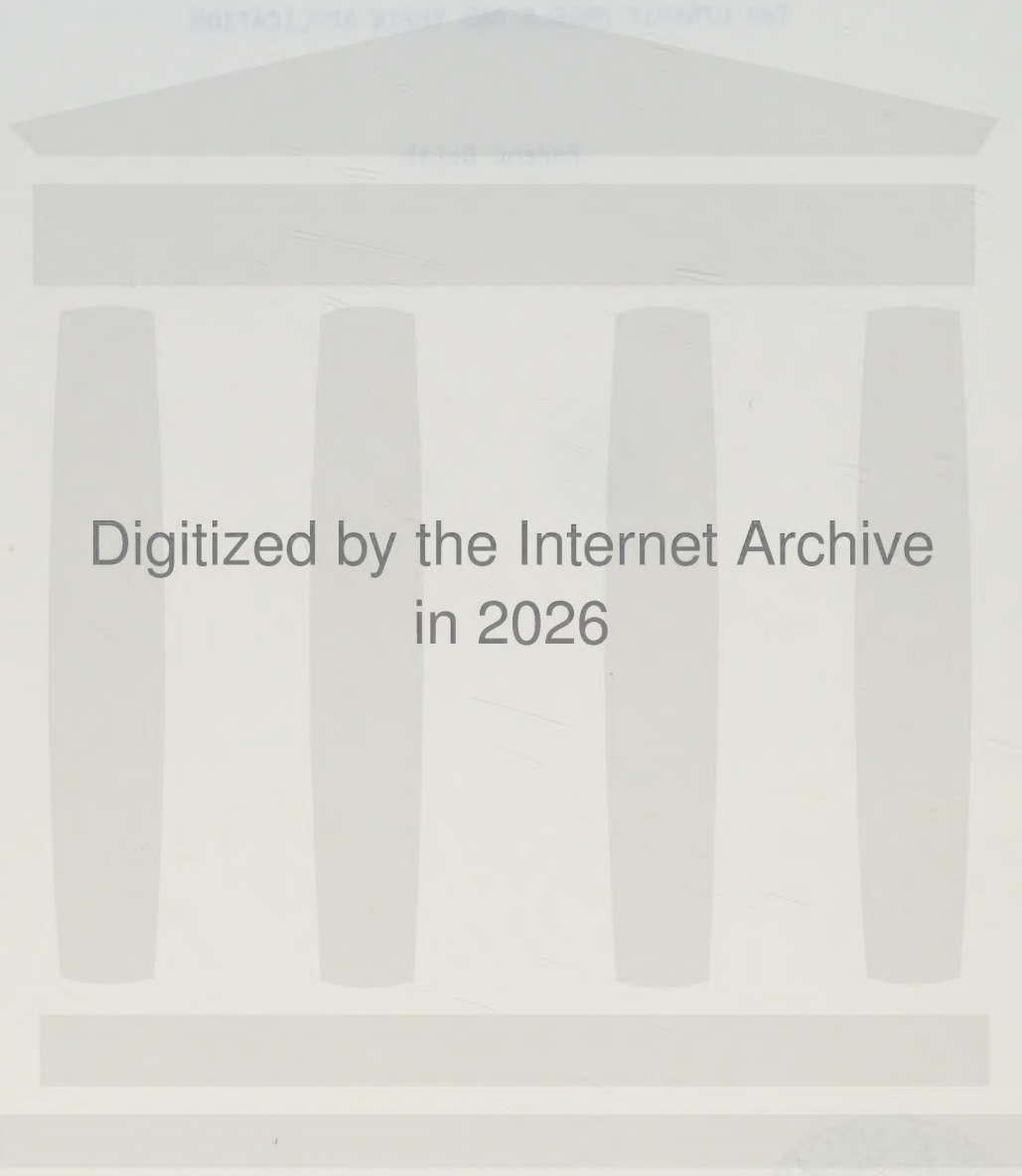
Ferenc Belik

A set model is developed by means of state-valued logic, making it possible to describe conditions in elements which act as the nodes of a tree or set. In application of the model to relational algebra, it is shown that the applicability of a relational data base query can be used of the model. Another model, a graph model, is developed based on a way of representing cyclic graphs. The use of this model improves considerably the way of solving the problem of searching for solutions in resource allocation systems. Furthermore, it gives a very efficient solution of problems which arise in solving resource allocation problems. In conclusion, the model is applied to a resource allocation problem. In summary, the model's applicability is also presented.



LUND UNIVERSITY

Department of Computer Sciences, May 1986



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41546727_0070

Abstract

A set model is developed by means of three-valued logic, making it possible to perform operations on elements which are on the move to or from a set. An application of the model to relational algebra shows that the accessibility of a relational data base increases by use of the model. Another model, a graph model, is developed based on a new way of representing directed acyclic graphs. The use of this model improves considerably on the running time of algorithms for deadlock avoidance in resource allocation systems. Furthermore, it gives a very efficient solution of deadlock avoidance problem in fully distributed environments. As an intermediate product, an improvement of a resource allocation algorithm, the banker's algorithm, is also presented.

Acknowledgements

I am grateful to Dr Eric Astor, Dr Bengt G. Lundberg and Svante Carlsson for their encouragement and useful suggestions. I would like especially to thank Dr Rolf Karlsson for his stimulating lectures on algorithm theory, for his encouragement and useful suggestions.

Table of Contents

INTRODUCTION	1
PART I: A Symbolic Set Model	3
Description of a Symbolic Set Model	5
1. Introduction	5
2. Operations on Elements in the Model	7
2.1 Arithmetic Comparison Operations	7
2.2 The Set Operator In memory of my mother	9
3. Locking	12
3.1 Locking at Record Level	12
3.2 Locking at the Lowest Level	16
3.3 Locking of Values: "Null-Values"	18
4. An Additional Remark	20
Appendix A: Set List	23
Appendix B: The Relational Algebra Operations	25
Acknowledgements	29
References	30
PART II: An Intermediate Product	31
Description of the Intermediate Product	33
1. Introduction	33
2. The Modified Dantzig's Algorithm	34
3. Simulation	38
4. Conclusion	47
Acknowledgements	49
References	50
PART III: A Dynamic Graph Model	51
Description of the Dynamic Graph Model	53
1. Introduction	53
2. The Dynamic Technique	57

Table of Contents

INTRODUCTION	1
PART I: A Dynamic Set Model	3
<u>Towards a Dynamic Set Model</u>	5
1. Introduction	5
2. Operations on Elements in the Model	7
2.1 Arithmetic Comparison Operations	7
2.2 The Set Operations $\cap$, $\cup$ and $-$	9
3. Locking	12
3.1 Locking at Record Level	12
3.2 Locking at the Lowest Level	16
3.3 Locking of Values: "Null-Values"	18
4. An Additional Remark	20
Appendix A: Set Laws	20
Appendix B: The Relational Algebra Operations	25
Acknowledgement	29
References	30
PART II: An Intermediate Product	31
<u>Deadlock Avoidance with a Modified Banker's Algorithm</u>	33
1. Introduction	33
2. The Modified Banker's Algorithm	34
3. Simulation	46
4. Conclusion	49
Acknowledgements	49
References	50
PART III: A Dynamic Graph Model	53
<u>Dynamic Operations on Directed Acyclic Graphs</u>	55
1. Introduction	56
2. The Dynamic Technique	57

3. Comparison between the Dynamic Technique and Topological Sorting	66
4. Conclusion	69
Acknowledgements	70
Appendix: Algorithm for Determining from the Path Matrix whether an Edge (v,w) Exists	70
References	72

<u>An Efficient Deadlock Avoidance Technique</u>	73
1. Introduction	74
2. Definitions and Representation of an Acyclic Digraph	75
3. Deadlock Avoidance	79
3.1 The Resource Allocation Graph and its Representation ...	80
3.2 The Dynamic Technique Used in the Resource Allocation Algorithm	84
3.3 Deletion and Insertion of Edges	89
3.4 Deletion and Insertion of Nodes	92
3.5 Attempted Insertion of Nodes and Edges	98
4. Conclusion	99
Acknowledgement	100
References	101

<u>An Efficient Technique to Avoid Deadlock in Distributed Systems</u>	103
1. Introduction	104
2. Definitions and Representation of an Acyclic Digraph	106
3. Deadlock Avoidance	110
3.1 The Resource Allocation Graph and its Representation ...	110
3.2 The Dynamic Technique Used in the Resource Allocation Algorithm	115
3.3 Deletion and Insertion of Edges	120
3.4 Deletion and Insertion of Nodes	121
4. Application of the Technique to Distributed Systems	124
4.1 Centralized Approach	125
4.2 Fully Distributed Approach	126
4.2.1 Introduction	126
4.2.2 The Modified Mutual Exclusion Algorithm	128

4.2.3 Allocation and Release of Resources to/from Processes	131
4.2.4 Allocation and Release of Processes and Resources to/from the System	141
4.2.5 Unexpected Release of Processes and Resources	145
5. Conclusion	146
Acknowledgement	147
References	148
CONCLUSIONS AND FURTHER RESEARCH	151

INTRODUCTION

At an abstract level, data processing can be considered as movement of data between sets. The initial idea of this thesis work was to construct a dynamic model where movement of data can be expressed, and then, to apply the model to several types of data processing.

We first developed a dynamic set model by means of three-valued logic, making it possible to perform operations on elements which are on the move to or from the set. We have applied the model to data bases where the transactions: updating, deletion and insertion correspond to movement of data. These transactions also demand locking of data; it must not occur, for example, that an airline reservation system allows selling of the same seat twice. We have shown that the accessibility of data in a relational data base can be increased by means of the model. This can be achieved in our model by performing operations on relations with locked elements. Since relational algebra operations can lead to elimination of locked elements we obtain immediately useful results without waiting for the lockings to be released. The dynamic model and its application to relational algebra is presented in Part I.

Next, when we tried to apply the dynamic set model to a resource allocation system with several instances of a resource type, we found an improvement of an resource allocation algorithm, called banker's algorithm. Banker's algorithm has a running time $O(mn^2)$, where m is the number of resource types and n is the number of processes. Our modification of the algorithm, given in Part II, has a running time of $O(mn)$ under certain conditions which are prevalent in most cases. But in the worst case, it still has an $O(mn^2)$ running time.

The application of the model to a resource allocation system with a single instance of each resource type gave fast algorithms which however were restrictive, in the sense that they rejected alloca-

tions of resources even though the allocations would not put the system in an unsafe state. We do not present these algorithms here. To improve these algorithms we made several attempts which finally led to a dynamic_graph_model (here, the word 'dynamic' has another meaning than in the set model). This model is based on a new way of representing directed acyclic graphs. We use a path matrix representation which makes it possible to efficiently detect cycles when tempting to insert edges. The model can be very efficiently used for deadlock avoidance in resource allocation systems, where indeed a rejected request of a resource is the most frequent operation according to simulation result. The cost for this operation is $O(1)$ by the use of the model. This $O(1)$ cost of a rejected request can then amortize the cost of the other types of operation and linear (or even constant) amortized time for an operation is realistic in resource allocation systems. By amortized time we mean the time of an operation averaged over a worst-case sequence of operations. A comparison with topological sorting, which is the previously proposed technique for detection of deadlocks, when the number of edges is $\theta(n^2)$, shows that our technique is superior. It is better by an additive term when allocating a resource and it is much better when rejecting a resource request, requiring only $O(1)$ time while topological sorting needs $\theta(n^2)$. We give the model in Part III, where also the comparison with topological sorting is presented. Then, the application of the model to tightly coupled multiprocessor systems is presented, and finally, the application of the model to distributed systems is given. We have thereby achieved considerable improvements of the running time of deadlock avoidance algorithms. Furthermore, we present a very efficient solution to the deadlock avoidance problem in a fully distributed environment. Since the presentation has the form of reports certain parts of the text recur in Part III.

PART I: A Dynamic Set Model

T O W A R D S A D Y N A M I C S E T M O D E L

Ferenc Belik

Department of Computer Science

Lund University, Sweden

Abstract

An attempt is made to represent motion conditions of elements of a set with support of three-valued logic. The use of arithmetic comparison operations and the traditional set operations is described. It is shown that the usual set laws are also valid for the model, with some modification. The model makes it possible to obtain immediately useful results from relations with locked parts in a relational data base. A semantic aspect of the model is illustrated by a draft of an interpretation of "null-values".

1. Introduction

A set is usually considered as a static object on which operation can be performed, sometimes several operations at the same time. A typical example is an on-line transaction system working on a data base. If an operation is expected to alter the set, then the set (or a subset thereof) must be locked during the operation. This is done for security reasons; it must not occur, for example, that an airline reservation system allows selling of the same seat twice.

Locking means the static condition of the set is maintained, and consequently all operations work on sets that are static objects. In fact, in two-valued logic it is not possible to express the dynamics during the operations. We can only express that an element belongs to a set or that it does not since the

two-valued logic has only two truth values: true and false.

An attempt is here made to express the dynamics with support of three-valued logic, by the truth values: true (T), maybe (M) and false (F). The truth value maybe is used for marking elements which are involved in operations, i.e. they can be removed from or inserted into the set. It marks an uncertain set membership.

The three-valued logic used here, taken from the Kalman-implication, has been inter alia used to express operations on "null-values" in a relational data base (Codd 1975). The soundness and the completeness of the Kalman-implications can be proved (Makinson 1973), we here give only the truth value tables:

$\alpha \quad ! \quad \neg \alpha$ ---!---	$\wedge \quad ! \quad T \quad M \quad F$ ---!-----	$\vee \quad ! \quad T \quad M \quad F$ ---!-----
T ! F	T ! T M F	T ! T T T
M ! M	M ! M M F	M ! T M M
F ! T	F ! F F F	F ! T M F
Negation	Conjunction	Disjunction

The Kalman-implication works like the classical two-valued logic when just the truth values true and false occur in the operations. This makes it possible to do an implementation where the truth value tables for three-valued logic are used throughout.

We can construct a model with the aid of this three-valued logic. To illustrate the dynamics, a metaphor can be used. Assume that a set corresponds to a number of people which are in a house and which pass the hall when they enter or leave the house. When we meet somebody in the hall with the overcoat on, then we don't know if the person is on the way out of the house or has just arrived. We can imagine a set in the same way with an associated hall and elements passing the hall on the way into or out of the set. If an element is

found in the hall, then we don't know if the element will belong to the set or not, and the truth value for the element's set membership is maybe.

2. Operations on elements in the model

To be able to describe operations on elements which are in the set or in the hall we can use the convention that we add a mark to every element. A mark has two values: indoors and in_the_hall; corresponding to the truth values true and maybe.

Example: $\{(a, \text{indoors}), (b, \text{indoors}), (c, \text{in_the_hall})\}$

We consider a set consisting of 0 or more elements where an element mark only indicates the set membership of the element.

2.1 Arithmetic comparison operations

The result of arithmetic comparison operations on elements may have three truth values; true and false as usual, and maybe if the comparison is true but one or both elements are in the hall.

Example: $(3, \text{indoors}) < (4, \text{in_the_hall})$ is maybe and
 $(3, \text{indoors}) = (4, \text{in_the_hall})$ is false.

The semantic meaning of maybe is the same as for the English word maybe: the result of the comparison can be true or false depending on the outcome of the operations which affect the element's set membership.

The arithmetic comparison operations get a new nuance here: they are coupled with set membership. We obtain the truth values:

true if the comparison is true and both elements are in the set

maybe if the comparison is true and one or both elements are in the hall of the set

false if the comparison is false or an element has left the hall (and the set).

Remark: We get the result false when comparing an element which has left the hall (and the set) as a consequence of the semantic meaning of the truth value maybe, and has the implication that the existence of an element is bound to the current set. Maybe means, as we recall, that the comparison is true but one or both elements are on the way into or out of the set. The dynamics is expressed in that if the element enters the set at the next time then we obtain the truth value true, and if the element exits the hall (and the set) at the next time then we should obtain another truth value. This latter truth value has to be false since the two other truth values (true and maybe) are already occupied. The case of false cannot be obtained as a result of operations but is a consequence of the interpretation of maybe.

By the above argument, we get the truth value false as the result of the comparison with a nonexisting element. This is of interest if we compare this model with the model which considers the "null-values" in a relational data base. "Null-values" may be regarded as elements in "the hall" of the set of elements with defined values, and not as nonexisting elements (see in section 3.3) .

With the computation of a comparison, the truth value table for conjunction is used. The marking values indoors and in_the_hall are taken as the truth values true and maybe.

Example: The result that $(3, \text{indoors}) < (4, \text{in_the_hall})$ is maybe is obtained in the following manner:

$$\begin{array}{c}
 (3 < 4) \wedge \text{indoors} \wedge \text{in_the_hall} \\
 \underbrace{\quad \quad \quad}_{\text{true}} \\
 \underbrace{\quad \quad \quad}_{\text{true}} \\
 \underbrace{\quad \quad \quad}_{\text{maybe}}
 \end{array}$$

2.2 The set operations $\cap$, $\cup$ and $-$

The set operations work as usual when both elements are marked indoors. If one or both elements are marked in_the_hall then we have:

$\cap$ If one or both elements are in the hall then we pick the element with the mark in_the_hall for the result set.

Remark: This corresponds to the operation $(\text{indoors} \wedge \text{in_the_hall})$ in the three-valued logic with the result in_the_hall. The semantic meaning is that we are not sure whether the element will belong to the set or not.

$\cup$ i) If both elements are marked in_the_hall then we pick one element for the result set according to the usual definition of a set.

ii) If one of the elements are marked in_the_hall and the other indoors then we pick the element with the mark indoors for the result set.

Remark: We get the result in analogy with the operation $\vee$ in three-valued logic.

In case ii) we pick the element which is "indoors" in a natural way since this element would belong to the result set in any case. Excluding the element which is in the hall means that we alter the time order in some meaning, as if the element already had entered or left the set via the hall. In other word, the operation carries out as much as possible, like the operations $\cap$ and $-$.

To allow that the same element can be found both in the set and in the hall at the same time would be in analogy with the intermediate status of the hall with respect to

set membership (see the complement laws in Appendix A), but it would also have the consequence that the distributive laws would not be valid, which is not desirable.

Example: $A = \{(a, \text{indoors})\}$; $B = \{(a, \text{in_the_hall})\}$

$$A \cup (B \cap A) \neq (A \cup B) \cap (A \cup A) \quad \text{for} \\ \{(a, \text{indoors}), (a, \text{in_the_hall})\} \neq \{(a, \text{indoors})\}$$

— Regard the expression $A - B$.

- i) If the A-element has the mark indoors and the B-element in_the_hall then we pick the element with the mark in_the_hall for the result set.
- ii) If the A-element has the mark in_the_hall and the B-element indoors then we pick none of the elements for the result set.
- iii) If both the A- and B-elements are marked in_the_hall then we pick one of the elements with the mark in_the_hall for the result set.

Remark: The semantic meaning of case i) is that if the B-element would leave the hall (and the set) then the element would belong to the result set, i.e. the set membership of the element is uncertain. The semantic meaning of case ii) is that whatever the set membership of the A-element is changed to, the element cannot belong to the result set. The semantic meaning of case iii) is that if the A-element enters the set and the B-element leaves the hall (and the set), then the element will belong to the result set. Otherwise, the element will not belong to the result set. It has as a consequence that the identity $A - (A - B) = A \cap B$ is not valid.

Another alternative, which preserves the intuitive meaning of the difference operation in case iii) and for which the identity $A - (A - B) = A \cap B$ is valid, is that the result set for intersection also contains elements which are in the hall of only one of the sets. This fits well with the identity

$$A \cap A' = \square_A \quad (\text{see Appendix A}), \text{ and has the semantic}$$

meaning that as soon as two sets are connected to each other by an operation, their hall becomes common. This is an interesting alternative but we choose the "more traditional" alternative as described above.

Example:

$$A = \{(a, \text{indoors}), (b, \text{in_the_hall}), (c, \text{indoors}), (d, \text{in_the_hall})\}$$

$$B = \{(a, \text{in_the_hall}), (b, \text{in_the_hall}), (c, \text{indoors})\}$$

$$A \cap B = \{(a, \text{in_the_hall}), (b, \text{in_the_hall}), (c, \text{indoors})\}$$

$$A \cup B = \{(a, \text{indoors}), (b, \text{in_the_hall}), (c, \text{indoors}), (d, \text{in_the_hall})\}$$

$$A - B = \{(a, \text{indoors}), (d, \text{in_the_hall})\}$$

$$B - A = \{(b, \text{in_the_hall})\}$$

$$A - (A - B) = \{(a, \text{in_the_hall}), (b, \text{in_the_hall}), (c, \text{indoors}), (d, \text{in_the_hall})\}$$

The computation of the result sets is done with the use of the truth value tables of the Kalman-implication.

Remark: We see in the example above that $A \cap B = B$, which means that $B \subseteq A$. But we have to observe that the result of the inclusion operation may have two values: true and maybe. The result maybe is obtained when $A \cap B$ contains some element in the hall.

In Appendix A it is shown that the usual set laws are also valid for this model, with some modification. Furthermore, the model works for static sets as usual with two-valued logic.

3. Locking

The model makes computation possible on elements which are on the move to or from a set. A practical use of the model arises with transactions on a data base where we can use the mark in_the_hall to mark locking of an element of the set. One can have various degrees of granularity. A finer granularity of locking allows a higher degree of accessibility at a greater cost in managing the locks. The relation between the degree of granularity and the degree of accessibility has been treated in Ries 1977 and 1979 .

3.1 Locking at record level

Assume that we have a relational data base and that we have time consuming transactions. Assume further that we want a high degree of accessibility. The degree of accessibility is increased if we are able to operate on entire relations containing locked parts, and get immediately useful results.

We can get an increase of accessibility through the operations: intersection, union, difference, projection, selection, -join, natural join and quotient; that is, all operations except Cartesian product can lead to elimination of the locked tuples. In this way, because we don't need to wait for the locking to be removed, we can obtain immediately useful results (i.e. relations without locked tuples) of operations on relations which contain locked tuples.

The application of the model on relational algebra is described in Appendix B. To mark locking of tuples we will use the notations T (true) and M (maybe) instead of the less economical notations indoors and in_the_hall.

Example: Intersection and difference

R:-----	S:-----
a b c T	a b c T
b c d T	e f g M
R ∩ S:-----	
a b c T	
R - S:-----	
b c d T	

Example: Union and projection

R: A B C	S: A B C
-----	-----
a b c T	a b c T
b c d M	b c d T
a c d T	
R ∪ S: A B C	

a b c T	
b c d T	
a c d T	
π(R): B C	
B,C -----	
b c T	
c d T	

Example: Selection and θ -join

R: A B C	S: D E
-----	-----
1 2 3 T	1 8 T
4 5 6 M	5 9 M
6 7 8 T	

$\sigma(R): A\ B\ C$
A=1 $\vee$ -----
C=8 1 2 3 T
 6 7 8 T

$R \bowtie S: A\ B\ C\ D\ E$
A=D $\vee$ -----
 $E \leq C$ 1 2 3 1 8 T
 6 7 8 1 8 T

Example: Natural-join

R: A B C D	S: C D E
-----	-----
a b c d T	c d e T
e f c d T	c d i T
e f g h T	g h i T
i j j j M	i j k T

$R \bowtie S: A\ B\ C\ D\ E$

a b c d e T
a b c d i T
e f c d e T
e f c d i T
e f g h i T

Example: Quotient

R:-----	S:-----
a b c d T	c d T
a b e f T	e f T
b c e f M	
e d c d T	
e d e f T	
a b d e M	

$R \div S:-----$
a b T
e a T

When we obtain only T-tuples, the result is immediately useful in the sense that it is correct at this point of time. The possibilities at the next point of time is not included in the result.

Immediately useful results concerning tuples which are locked can be obtained only as negative information. For example, we have the following query while the "machine bolt" are updated:

$$\exists x(\text{Quantity}(\text{machine_bolt}, x) \wedge x \geq 150)$$

If $x \geq 150$ then we obtain the answer maybe which means that at least 150 machine bolts are found at this point of time, but it is uncertain if they are on the way into or out of the set, and accordingly the result is not immediately useful.

If $x < 150$ then we get the answer false which means that less than 150 machine bolts are found at this point of time. And despite the uncertainty if they will enter or leave the set, this result is immediately useful, since in any case we do not have 150 or more machine bolts at this point of time.

The implementation of locking can be arranged so that the tuples which are being altered are put in the hall (in a physical data space for each relation). If we look for tuples first in the hall of the relation and then in the relation itself, it is equivalent to the marking of tuples.

Thus the model makes it possible to increase the accessibility of a relational data base which has time consuming operations. For this increased accessibility we have to pay in terms of marking at tuple level and in terms of the use of three-valued logic instead of two-valued logic.

Observe that for the implementation of the three-valued logic, the available two-valued logic must be used. Alternatively, the entire model could be implemented by two-valued logic, but

this put demands on the programmer to make all the considerations we have done in section 2. Furthermore, instead of using the truth value tables for the Kalman-implication, the programmer would have to combine the truth values of two-valued logic operations in if- or case-statements. We can say that three-valued logic lifts up the model one level and in this way makes the model simpler, hiding the basic two valued logic.

3.2 Locking at the lowest level

If we make the locking granularity finer, at the level of attribute values, then the mark M on a tuple means that some attribute values of the tuple are "in_the_hall" (of the tuple; tuples are regarded as sets). We can work with these tuples in the same way as described above. But this finer granularity of marking increases the accessibility only if we can obtain T-tuples from M-tuples by some operations. We want operations which can eliminate attribute values (columns in relations). These are only the projection, union and natural-join operations.

Example: Projection

R: A	B	C	D		S:
-----					-----
(a,T)	(b,T)	(c,T)	(d,T)	T	(c,T) (d,T) T
(a,T)	(b,T)	(e,T)	(f,T)	T	(e,T) (f,T) T
(b,T)	(c,T)	(e,T)	(f,M)	M	
(e,T)	(d,T)	(c,T)	(d,T)	T	
(e,T)	(d,T)	(e,T)	(f,T)	T	
(a,T)	(b,M)	(d,T)	(e,T)	M	

$\pi(R):$

A	C
(a,T)	(c,T) T
(a,T)	(e,T) T
(b,T)	(e,T) T
(e,T)	(c,T) T
(e,T)	(e,T) T
(a,T)	(d,T) T

Example: Union

R:	S:
(a,T) (b,T) (c,T) T	(a,T) (b,T) (c,T) T
(b,M) (c,T) (d,T) M	(b,T) (c,M) (d,T) M

R $\cup$ S:
(a,T) (b,T) (c,T) T
(b,T) (c,T) (d,T) T

Example: Natural-join

Here the union rule is used with the joining of matching columns.

R:	A	B	C

(a,T)	(b,M)	(c,T)	M
(e,T)	(b,T)	(c,T)	T

S:	B	C	D

(b,T)	(c,M)	(d,T)	M
(b,T)	(c,M)	(e,T)	M

R $\bowtie$ S:	A	B	C	D

(a,T)	(b,T)	(c,T)	(d,T)	T
(a,T)	(b,T)	(c,T)	(e,T)	T
(e,T)	(b,T)	(c,T)	(d,T)	T
(e,T)	(b,T)	(c,T)	(e,T)	T

Of course the elimination of tuples also leads to elimination of attribute values.

Example: R and S given in the example for projection.

R $\div$ S: _____
 (a,T) (b,T) T
 (e,T) (d,T) T

Since only these operations can explicitly lead to elimination of locked elements, this means that no greater profit can be achieved here except when projection, union and natural-join are the most common operations.

3.3 Locking of values: "null-values"

If we change the point of view and regard the model as a set of existing elements with values, then "null-values" (a special case of incomplete information with the meaning "attributes applicable but values are unknown", see Biskup 1983, Codd 1975, 1979, and Gallaire 1984) can be regarded as locked elements, i.e. elements which are in the hall.

The result of arithmetic comparison operations is defined maybe when one of the two elements is found in the hall, otherwise (if both elements are in the set) it is defined as in two valued logic.

Codd's model for "null-values", based on three-valued logic (with the "unknown" interpretation), has been criticized for, among other things, that it does not produce the correct answer for queries which correspond to tautologies (Zaniolo 1982,1984).

Considering the problem with tautology from our model's viewpoint, it turns out to be less problematic. Consider the following expression type, where the age of John is unknown (i.e. a "null-value"):

$$\exists x(\text{Age}(\text{John}, x) \wedge x \leq 60) \vee (\text{Age}(\text{John}, x) \wedge x > 60)$$

The existence of the age of John is bound to the set; we have a kind of "closed-world" situation (Reiter 1978). Since we know that John exists and therefore that his age also must exist, we expect the truth value true. But we obtain the truth value maybe, which is correct in this situation, meaning that the age of John is locked (i.e. unknown).

In the same way the truth value maybe is a correct result for the following expression:

$$\exists y(\text{Telephonenumber}(\text{John}, y) \wedge y \leq 263400000) \vee (\text{Telephonenumber}(\text{John}, y) \wedge y > 263400000)$$

Here we do not have the same intuitive expectation as in the previous example to get the truth value true since the existence of John's telephone number is not as certain as the existence of his age.

These examples show that the semantic meaning of tautology is changed with the use of three-valued logic, since also the truth value maybe can be obtained. This because the introduction of the third truth value makes Aristotle's law of the excluded third inappropriate, i.e. every statement is not necessarily true or false.

The implementation gives a dynamic image of "null-values". If we obtain positive information, i.e. the "null-value" gets a value, then the element enters the set and we get the result true for the query expressions above. If we do not obtain any information then we get the result maybe, and if we obtain negative information, i.e. that the value does not exist (as it can be with John's telephone number) then the element exits the hall (and the set) and we get the result false.

The implementation draft also shows that it is inappropriate to regard a "null-value" as a nonexisting element, since we

are not able to work with such elements (they lie outside the set and the hall). If we want "null-values" also to get the meaning "value does not exist", then we have to use the over-all meaning "no information" suggested by Zaniolo.

4. An additional remark

One can say that this model is dynamic in the sense that it registers motion conditions of an element. But on the other hand one could also view the motion conditions as frozen. To make the model more dynamic one could mark the direction of motion of elements which are in the hall (as "entering the set" or "exiting the set") .

APPENDIX A

Set laws

The set laws for static sets are valid here with that modification that the empty set and the universal set have an extended meaning.

The commutative laws

The laws $A \cup B = B \cup A$ and $A \cap B = B \cap A$ are valid because of the symmetry of the Kalman-implication's truth value tables.

The identity laws

The laws $A \cup \emptyset = \emptyset \cup A = A$ and $A \cap U = U \cap A = A$ are valid but get an altered meaning; see below at the complement laws.

The complement laws

To have the laws $A \cup A' = U$ and $A \cap A' = \emptyset$ valid while letting the hall have an intermediate position with respect to the set membership, we can let the elements in the hall of A be found also in the hall of A' . The consequence is that the empty set loses its universal character and will depend on the set it is connected to (via an operator). It becomes identical with the hall of the set. This meaning of $\emptyset$ works well for the identity law $A \cup \emptyset = \emptyset \cup A = A$.

We have to observe that with only static sets, i.e. sets with empty halls, the usual meaning of the empty set is valid. The result of $A \cap B$ where the sets A and B have no common elements, (not in their halls either) is the "usual" empty set.

To let A have the same hall as A' has another consequence too. The law $A \cup A' = U$ implies that the universal set consists of not only static elements (with mark indoors) but also of elements which are on the move (with mark in_the_hall). This because of the rules for union. The universal set is regarded as a set of elements which can be both static and mobile.

Example: If $A = \{(a, \text{indoors})\}$ and $A' = \{(b, \text{indoors})\}$ then $U = \{(a, \text{indoors}), (b, \text{indoors})\}$. If $A = \{(a, \text{in_the_hall})\}$ and $A' = \{(a, \text{in_the_hall}), (b, \text{indoors})\}$ then $U = \{(a, \text{in_the_hall}), (b, \text{indoors})\}$.

This meaning of the universal set works well for the other identity law $A \cap U = U \cap A = A$.

Addendum:

To make clear the extended meaning of the empty set, we can introduce the notation $\square_A$ for the hall of the set A . The intermediate position of the hall means that $\square_A \subseteq A$ and $\square_A \subseteq A'$.

We obtain that

i) $A \cap A' = \emptyset$, $\emptyset' = U$ and $A \cup \emptyset = A$ if the hall to A is empty

ii) $A \cap A' = \square_A$, $\square'_A = \square_A$ and $A \cup \square_A = A$ if some element is found in the hall to A

We further obtain from the identity law that

$$\square_A = \square_A \cup \emptyset = \emptyset$$

We should observe that the cases i) and ii) are manifesting different time situations and that this latest identity cannot occur if $\square_A$ is not empty. Thus the identity $\square_A = \emptyset$ only means that $\square_A$ and $\emptyset$ are identical in the sense of "a hall".

One can say that the dynamics of the model is expressed in the extended meaning of the empty set and the universal set.

The associative laws

To see that the laws $A \cup (B \cap C) = (A \cup B) \cap C$ and $A \cap (B \cup C) = (A \cap B) \cup C$ are valid it is sufficient to examine the cases when all three sets consist of one and the same element with only the element's set membership varying. We get 8 (2×3) cases where it is enough to work with the element marks.

A	B	C	$A \cup (B \cup C)$	$(A \cup B) \cup C$
indoors	indoors	indoors	indoors	indoors
in_the_hall	indoors	indoors	} indoors	} indoors
indoors	in_the_hall	indoors		
indoors	indoors	in_the_hall		
in_the_hall	in_the_hall	indoors	} indoors	} indoors
in_the_hall	indoors	in_the_hall		
indoors	in_the_hall	in_the_hall		
in_the_hall	in_the_hall	in_the_hall	in_the_hall	in_the_hall

and

A	B	C	$A \cap (B \cap C)$	$(A \cap B) \cap C$
indoors	indoors	indoors	indoors	indoors
in_the_hall	indoors	indoors	} in_the_hall	} in_the_hall
indoors	in_the_hall	indoors		
indoors	indoors	in_the_hall		
in_the_hall	in_the_hall	indoors	} in_the_hall	} in_the_hall
in_the_hall	indoors	in_the_hall		
indoors	in_the_hall	in_the_hall		
in_the_hall	in_the_hall	in_the_hall	in_the_hall	in_the_hall

We see that the associative laws are valid. We get the result indoors using the union operation when one element is "indoors". In the same way, we get the result in_the_hall using intersection. This accords with the operations $\cup$ and $\cap$ in the Kalman-implication.

The distributive Laws

In the same manner we can verify that the laws $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ and $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ are valid.

A	B	C	$A \cup (B \cap C)$	$(A \cup B) \cap (A \cup C)$
indoors	indoors	indoors	indoors	indoors
in_the_hall	indoors	indoors	indoors	indoors
indoors	in_the_hall	indoors	indoors	indoors
indoors	indoors	in_the_hall	indoors	indoors
in_the_hall	in_the_hall	indoors	in_the_hall	in_the_hall
in_the_hall	indoors	in_the_hall	in_the_hall	in_the_hall
indoors	in_the_hall	in_the_hall	indoors	indoors
in_the_hall	in_the_hall	in_the_hall	in_the_hall	in_the_hall

and

A	B	C	$A \cap (B \cup C)$	$(A \cap B) \cup (A \cap C)$
indoors	indoors	indoors	indoors	indoors
in_the_hall	indoors	indoors	in_the_hall	in_the_hall
indoors	in_the_hall	indoors	indoors	indoors
indoors	indoors	in_the_hall	indoors	indoors
in_the_hall	in_the_hall	indoors	in_the_hall	in_the_hall
in_the_hall	indoors	in_the_hall	in_the_hall	in_the_hall
indoors	in_the_hall	in_the_hall	in_the_hall	in_the_hall
in_the_hall	in_the_hall	in_the_hall	in_the_hall	in_the_hall

We see that the distributive laws are valid.

Since these set laws are valid we can work with the set operations "as usual". Furthermore, we should observe that the classical set operations are always valid for static sets. This can be seen, for example, with the associative and distributive laws in the first rows in their respective tables.

APPENDIX B

The relational algebra operations

The set model with a hall can be applied to the relational algebra where a set corresponds to a relation and an element corresponds to a tuple. The notations T (true) and M (maybe) are used below instead of the less economical notations "in-doors" and "in_the_hall" used in Appendix A .

The operations $\cap$, $\cup$ and $-$

These operations works in the same way as described in section 2.2 .

Example: R:_____	S:_____
a b c T	a b c M
b c d M	b c d M
c d d T	c d d T
	d e f T

R $\cap$ S:_____

a b c M
b c d M
c d d T

R $\cup$ S:_____

a b c T
b c d M
c d d T
d e f T

R — S: _____
 a b c M
 b c d M

S — R: _____
 b c d M
 d e f T

Projection

With projection the result tuples have the same marks as they had before. To eliminate duplicates, the union rules are used.

Example: R: A B C

 a b c T
 b c d M
 c d d T

$\pi(R): A B$

A,C _____
 a c T
 b d M
 c d T

$\pi(R): C$

C _____
 c T
 d T

Selection

With selection the tuples for which the selection condition is true are selected and the tuples keep their marks.

Example: R: the same as with projection above.

$\sigma(R): A B C$

A=a $\vee$ _____
 B=d a b c T
 c d d T

Cartesian_product

With Cartesian product the mark will have the value M as soon as one of the two joined tuples has value M. This can be computed by means of the $\bigvee$ operator in three-valued logic.

Example : R: A B C

```
-----
a b c T
b c d M
c d d T
```

S: D E

```
-----
e f T
g e M
```

R $\times$ S: A B C D E

```
-----
a b c e f T
a b c g e M
b c d e f M
b c d g e M
c d d e f T
c d d g e M
```

Remark: The semantic meaning of the result mark M when one of the two joined tuples has value M is that if one half of the joined tuples is on the move into or out of the relation then the entire tuple cannot be considered to be "indoors".

 θ -join

If the θ -condition is true for the attribute values, then the joined tuples are picked for the result relation in the same manner as with the Cartesian product.

Example: R : A B C

```

-----
1 2 3 T
4 5 6 M
7 8 9 T

```

S: D E

```

-----
1 2 T
8 4 M

```

$R \bowtie S$: A B C D E

```

(A<E  $\wedge$  -----
C>E)  $\vee$  1 2 3 1 2 T
B=D      4 5 6 8 4 M
          7 8 9 8 4 M

```

Natural - join

The natural - join works in the same manner as the θ - join.

Example: R: A B C D

```

-----
a b c d T
e f c d T
e f g h T
i j i j M

```

S: C D E

```

-----
c d e T
c d i T
g h i M
i j k M

```

$R \bowtie S$: A B C D E

```

-----
a b c d e T
a b c d i T
e f c d e T
e f c d i T
e f g h i M
i j i j k M

```

Quotient

With the operation quotient a result tuple receives the mark M as soon as one of the involved tuples has mark M .

Example: R: _____ S: _____

a b c d M

a b e f T

b c e f M

e d c d T

e d e f T

a b d e T

c d T

e f T

R ÷ S: _____

a b M

e d T

Example: R: _____ S: _____

a b c d M

a b e f T

b c e f M

e d c d T

e d e f T

a b d e T

c d T

e f M

R ÷ S: _____

a b M

e d M

Acknowledgement

The author is grateful to Dr Bengt G. Lundberg for his useful suggestions.

REFERENCES

- Biskup 1983 Biskup, J. : A Foundation of Codd's Relational Maybe Operations, ACM TODS, Vol.8, No.4, Dec.1983, pp. 608-636.
- Codd 1975 Codd, E. F. : Understanding Relations, FDT 7:3, pp. 23-29, ACM New York, 1975.
- Codd 1979 Codd, E. F. : Extending the Database Relational Model to Capture More Meaning, ACM TODS, Vol.4, No.4, Dec.1979, pp.397-434.
- Gallaire 1984 Gallaire, H. & Minker, J. & Nicolas, J-M. : Logic and Databases: A Deductive Approach, ACM Computing Surveys, Vol.16, No.2, June 1984, pp. 153-185.
- Makinson 1973 Makinson, D. C. : Topics in Modern Logic, Methue & Co Ltd, London, 1973.
- Reiter 1978 Reiter, R. : On Closed World Data Bases, in Logic and Data Bases, Gallaire, H. & Minker, J., Eds, Plenum Press, New York, 1978.
- Ries 1977 Ries, D. R. & Stonebraker, M. : Effects of Locking Granularity in a Database Management System, ACM TODS, Vol.2, No.3, Sept.1977, pp. 233-246.
- Ries 1979 Ries, D. R. & Stonebraker, M. : Locking Granularity Revisited, ACM TODS, Vol.4, No.2, June 1979, pp. 210-227.
- Zaniolo 1982 Zaniolo, C. : Database Relations with Null Values, Principles of Database Systems Symp., Papers ACM-PODS, 1 March 1982.
- Zaniolo 1984 Zaniolo, C. : Database Relations with Null Values, Journal of Computer and System Sciences 28, 1984, pp. 142-166.

PART II: An Intermediate Product

DEADLOCK AVOIDANCE WITH A MODIFIED BANKER'S ALGORITHM

Ferenc Belik

Department of Computer Science

Lund University, Sweden

Abstract

In recent computer systems, deadlock has generally been viewed as a limited annoyance. Future computer systems will be oriented more towards asynchronous parallel operation and multiple processor systems will be common. In future systems, however, resource allocation will tend to be more dynamic and deadlock will be a far more critical consideration. The banker's algorithm, a deadlock-avoidance algorithm, provides a dynamic resource allocation but its running time is $O(mn^2)$. Our modification of banker's algorithm gives an $O(mn)$ running time under certain conditions which conditions are prevalent in most cases. But in the worst case, it gives an $O(mn^2)$ running time.

1. Introduction

A situation called deadlock was pointed out by Dijkstra [6] in a computer system in which two or more processes coexist sharing resources. A simple example is where a process P_1 holds a device A and requires device B to continue, but device B is already held by another process P_2 which requires A to continue. There is a circular wait here and it will not be resolved unless a "drastic" action, i.e., preemption, is taken. Informally, a deadlock is a situation in which there is a process whose progress is blocked forever owing to lack of resources, regardless of the future activity (e.g., release of resource unit) of unblocked processes. Different aspects of the deadlock problem have been investigated in the past [4,8-11,14,19],

and an interest in it has been revived in the database and distributed environment [1,2,7,13,15-17,20,21]. Two necessary and sufficient conditions for the absence of deadlock were established under the expedient allocation policy in [3]. Coffman et al. [4] classify approaches to dealing with deadlocks into three categories: detection and recovery, avoidance, and prevention. This paper considers a deadlock-avoidance algorithm, called the banker's algorithm. The banker's algorithm was developed for a single resource type by Dijkstra [6], and was extended to multiple resource types by Haberman [8].

In recent computer systems [5], deadlock has generally been viewed as a limited annoyance. Most systems implement the basic deadlock prevention methods suggested by Havender [9], and these methods seem to be satisfactory. In future systems, however, resource allocation will tend to be more dynamic and deadlock will be a far more critical consideration. The banker's algorithm provides a dynamic resource allocation but its running time is $O(mn^2)$ in a system consisting of n processes and m different types of resources. An $O(mn^{1.5})$ algorithm, employing a network technique, was presented by Kameda [12]. Our modification of banker's algorithm gives a $O(mn)$ running time under certain conditions which conditions are prevalent in most cases. But in the worst case its running time is $O(mn^2)$.

2. The modified banker's algorithm

To describe the algorithm, we need two definitions:

Definition.

A sequence of processes $\langle p_1, p_2, \dots, p_n \rangle$ is a safe sequence if for each p_i , the resource which p_i can still request can be satisfied by the available resources plus the resources held by all the p_j , with $j < i$. $\square$

In this situation, if the resources process p_i needs are not immediately available, p_i can wait until all p_j have finished, then p_i can obtain all of its needed resources, complete its designated

task, return all of its allocated resources, and then terminate. When p_i terminates, p_{i+1} can obtain its needed resources, and so on.

Definition

A system is in a safe state if there exists a safe sequence of the processes in the system. If there exists no such sequence then the state of the system is unsafe. $\square$

Following Haberman [8], we assume that the maximum resource requirement (i.e., claim) of each process is known in advance. It is assumed that within this limit each process can make requests for resources dynamically in an arbitrary sequence.

The main difference between banker's algorithm and our modified algorithm is the way of finding a safe sequence. At every resource allocation, banker's algorithm tries to find a complete safe sequence, consisting of all of the n processes, whereas the modified algorithm only tries to find a safe sequence of the necessary length. This length decreases during the resource allocation until some process completes its task and its released resources become available again. These sequences of decreasing length consists of

- those processes for which the amount of the available resources are not enough to terminate at the present resource allocation moment but they were enough at the previous resource allocation moment, and
- of the process which has requested resources at the present resource allocation moment and if the available resources are not enough to terminate the process.

The reason why the modified algorithm only tries to find a safe sequence of the necessary length may be intuitively clear by the following example. The correctness of the algorithm is proved later.

Example

Consider a system of 5 processes and of one resource class, consisting of 15 resources. Let A stands for the number of resources allocated to each process and N stands for the remaining need of each process. Underlining marks which process allocates resources at the current moment.

at moment	t_n	t_{n+1}	t_{n+2}	t_{n+3}
P	A N	A N	A N	A N
p_1	2 3 T	2 3 T	<u>3</u> <u>2</u> T	3 2 T
p_2	2 6 T	2 6 M	2 6	<u>3</u> <u>5</u> M
p_3	1 1 T	1 1 T	1 1 T	1 1 T
p_4	<u>2</u> <u>8</u> M	<u>4</u> <u>6</u> M	4 6	4 6
p_5	2 4 T	2 4 T	2 4 M	2 4
available	6	4	3	2
become				
available	13 (+2)	9 (+6)	7 (+2)	6 (+3)

All of the allocation states are safe at the moments $t_n, \dots, t_{n+3}$. We use the T (true) to mark a process if the remaining need of the process is not greater than the amount of currently available resources (we have a row for this amount of resources in the table above). These processes belong to a safe sequence and do not need to be checked. They can finish their task and their allocated resources will "become available" (we have a row for this amount of resources in the table above). We use the M (maybe) to mark a process if the remaining need of the process is greater than the amount of the currently available resources, but if it was not greater at the previous moment (that is, the process had the mark T) or the process has allocated resources at the current moment. Only these processes need to be checked if they form a safe sequence (and not always, not at moment t_{n+2} in the example). The reason, why it is enough to check only the M-processes, is that the sum of the "become available" resources and the resources allocated by the M-processes (given in brackets in the table) is at least as large as the "become available" resources at the previous moment. It means, that if the previous allocation state was safe then also the current allocation state is safe. $\square$

Let n be the number of processes in the system and m the number of resource classes. We will need the following data structures:

- Available. A vector of length m indicating the number of available resources of each type. If $Available[j] = k$, there are k instances of resource type r_j available.
- Max. A $n \times m$ matrix defining the maximum demand of each process. If $Max[i,j] = k$, then process p_i may request at most k instances of resource type r_j .
- Allocation. An $n \times m$ matrix defining the number of resources of each type currently allocated to each process. If $Allocation[i,j] = k$, then process p_i is currently allocated k instances of resource type r_j .
- Need. An $n \times m$ matrix indicating the remaining need of each process. If $Need[i,j] = k$, then process p_i may need k more instances of resource type r_j in order to complete its task. Note that $Need[i,j] = Max[i,j] - Allocation[i,j]$.
- State. A vector of length n indicating the state of each process. If $State[i] = T$ (terminatable), there are enough available resources process p_i may need in order to complete its task, i.e., $Need[i,j] \leq Available[j]$ for all j . If $State[i] = M$ (maybe), then $(Need[i,j] > Available[j] \text{ for some } j \text{ at moment } t_n) \text{ and } [(Need[i,j] \leq Available[j] \text{ for all } j \text{ at moment } t_{n-1}) \text{ or } (\text{process } p_i \text{ has allocated its resources at moment } t_n)]$, where t_n means the current resource allocation moment and t_{n-1} means the previous resource allocation moment. If $State[i] = S$ (safe, i.e., the process will be a member in a safe sequence if the "M" processes at the current resource allocation moment form a safe sequence), then process p_i is not the process which has requested resources at the current allocation moment and $State[i]$ had the mark M at some previous resource allocation moment.

The above data structures vary both in size and values as time progresses.

To simplify the presentation of the algorithm, let us establish some notation. Let X and Y be vectors of length n . We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \dots, n$. For example, if $X = (1, 7, 3, 2)$ and $Y = (0, 3, 2, 1)$, then $Y \leq X$. $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocation_i and Need_i respectively. Allocation_i , for example, specifies the resources currently allocated to process p_i .

The main algorithm

- I. Let Request_i be the request vector for process p_i . If $\text{Request}_i[j] = k$, then process p_i wants k instances of resource type r_j . When a request for resources is made by process p_i , the following actions are taken:
 1. If $\text{Request}_i > \text{Need}_i$ then we have an error. The process has requested more resources than it had declared as its maximum.
 2. If $\text{Request}_i > \text{Available}$ then p_i must wait.
 3. The system pretends to have allocated the requested resources to process p_i by modifying the state as follows.
 - a) $\text{Available} := \text{Available} - \text{Request}_i;$
 - b) $\text{Allocation}_i := \text{Allocation}_i + \text{Request}_i;$
 - c) $\text{Need}_i := \text{Need}_i - \text{Request}_i;$
 - d) IF $\text{Need}_i \leq \text{Available}$
 THEN
 BEGIN (* the allocation state is safe *)
 e) $\text{State}[i] := T;$
 f) FOR $p := 1$ TO n DO
 IF $(p \neq i) \text{ AND } (\text{State}[p] = T) \text{ AND } (\text{Need}_p > \text{Available})$
 THEN $\text{State}[p] := M$
 ELSE
 IF $(p \neq i) \text{ AND } (\text{State}[p] = M)$
 THEN $\text{State}[p] := S;$
 END
 ELSE

```

BEGIN                                (* the safety of the allocation *)
g)   State[i] := M;                  (* state must be checked      *)
h)   FOR p := 1 TO n DO
      IF (p <> i) AND (State[p] = T) AND (Needp > Available)
      THEN State[p] := M (* only these processes will be *)
      ELSE                (* checked by the safety alg.  *)
        IF (p <> i) AND (State[p] = M)
        THEN State[p] := S; (* these processes will not be *)
                               (*checked by the safety alg.  *)
i)   control by the safety algorithm if the allocation state
      is safe;
END;
```

If the resulting resource allocation state is safe, the transaction is completed and process p_i is allocated its resources. However, if the new state is unsafe, then p_i must wait for Request_i and the old resource allocation state is restored.

II. If process p_i has completed its task the following actions are taken:

```

Available := Available + Allocationi;
Allocationi := 0;
FOR p := 1 TO n DO
  IF Needp ≤ Available
  THEN State[p] := T;
```

Safety_algorithm

The algorithm for finding a safe sequence can be described as follows:

1. Let Work be a vector of length m and let Finish be a n x 2 matrix.
Initialize:

```

c := 1;
Work := Available;
FOR p := 1 TO n DO
  BEGIN
    IF State[p] = T
      THEN Work := Work + Allocationp;
    IF State[p] = M
      THEN
        BEGIN
          Finish[c,1] := p;
          Finish[c,2] := FALSE;
          c := c + 1;
        END
      END
  END
END

```

2. Find an i such that $i < c$ and:

- a) $\text{Finish}[i,2] = \text{FALSE}$, and
- b) $\text{Need}_{\text{Finish}[i,1]} \leq \text{Work}$.

If no such i exists, go to step 4.

3. $\text{Work} := \text{Work} + \text{Allocation}_{\text{Finish}[i,1]}$;
 $\text{Finish}[i,2] := \text{TRUE}$;
 go to step 2.

4. If $\text{Finish}[i,2] = \text{TRUE}$ for all $i < c$, then the system is in a safe state.

Example

Since the algorithm works with all resource classes in parallel it is enough to regard the allocation of one type of resources. Let $\text{Available} = 13$ at moment t_0 , that is when the system starts. P stands for processes, A and N stands for the matrices Allocation and Need respectively, and S stands for the vector State. Underlining marks which process allocates resources at the current moment.

at moment	t_1	t_2	t_3	t_4	t_5	t_6
P	A N S	A N S	A N S	A N S	A N S	A N S
p_1	<u>2_3</u> T	2 3 T	2 3 T	2 3 T	2 3 M	<u>3_2</u> M
p_2			<u>2_6</u> T	2 6 M	2 6 S	2 6 S
p_3		<u>1_1</u> T	1 1 T	1 1 T	1 1 T	1 1 T
p_4				<u>4_6</u> M	4 6 S	4 6 S
p_5					<u>2_4</u> M	2 4 S
Available	11	10	8	4	2	1

All of the allocation states are safe at the moments $t_1, t_2, \dots, t_6$. □

Theorem

The modified banker's algorithm guarantees that the system is always in a safe state.

Proof

The safety algorithm tries to find a safe sequence consisting only of those processes which have the mark M. It is easy to see that the safety algorithm works properly.

To prove the main algorithm we use induction over the moments $t_0, t_1, \dots, t_n$, where at every moment $t_j, j > 0$, resources are allocated to some process p_k . The algorithm says that at moment t_n it is enough to control the existence of a safe sequence only for those processes p_k which have the mark M in the vector State. The mark M means, we recall, that $(\text{Need}_k > \text{Available at moment } t_n)$ and $[(\text{Need}_k \leq \text{Available at moment } t_{n-1}) \text{ or } (\text{process } p_k \text{ has allocated its resources at moment } t_n)]$.

- A) At moment t_1 , process p_i is allocated its resources, $\text{State}[i] = T$, and the allocation state is safe according to line I.3.e).
At moment t_2 , process p_k is allocated its resources and there are six possible markings in the vector State:

	State[i]	State[k]	
i)	T		$i = k$
ii)	T	T	
iii)	T	M	
iv)	M		$i = k$
v)	M	T	
vi)	M	M	

Only in cases iii), iv) and vi) control is needed by the safety algorithm. It is straightforward to check that the algorithm works properly.

B) Assume the system is in a safe state at moment t_n .

Assume further the system pretends to have allocated the requested resources to process p_j at moment $t_n + 1$.

1. If $\text{State}[j] = T$, then the system is in a safe state, because there are enough immediately available resources for process p_j to finish its task, and when it terminates the state at moment t_n can obtain its released resources.
2. If $\text{State}[j] = M$, then the processes, marked with M in the vector State, must form a safe sequence according to line I.3.i). All the resources this sequence is able to release can be found in the vector Work at step 4. in the safety algorithm. Let $\text{Work}_{t_{n+1}}^{(4)}$ stand for this contents of the vector Work. Let $\text{Work}_{t_n}^{(1)}$ stand for the contents of the vector Work at the end of step 1. in the safety algorithm at moment t_n (i.e., resources held by "T" marked processes + Available at moment t_n).
The rules of the marking in the vector State give that

$$\text{Work}_{t_{n+1}}^{(4)} \geq \text{Work}_{t_n}^{(1)}$$

(because the processes to be controlled at moment $t_n + 1$ were marked T at moment t_n , possible with the exception of p_j).

This means that the safe sequence at moment $t_n + 1$ guarantees that it is enough available resources for the safe sequence at the previous moment t_n (and for the moment $t_n - 1$, and so on), that is, there exists a safe sequence which contains all the processes in the system.

If a process has completed its task then the actions in the part II of the main algorithm are taken. These actions make sure of maximum use of the resources without influencing the safety of the system. $\square$

Running time

In the worst case:

When the amount of available resources at every two moments is oscillating between maximum (leading to that all of the processes will be marked T) and zero (leading to that all of the processes will be marked M), the time complexity is $O(mn^2)$. This situation can happen if someone of the processes releases and receives all resources every other moment. Such a situation must be exceptional, but if it occurs our modified algorithm will work twice as fast as banker's algorithm (because the processes marked T are not checked by the safety algorithm).

Under condition C:

Definition of condition C:

- no process completes its task under q moments
(implying that its released resources will become available again at part II in the main algorithm, where already checked M-processes can be converted into M-processes again at subsequent moments),
and
- no process is rejected after checking by the safety algorithm under q moments
(that is, those at moment t_{n-1} already checked M-processes are not checked again at moment t_n), and
- $q \geq cn$, where n = the number of processes and c is a constant.

Theorem

Under condition C the modified banker's algorithm works in $O(mn)$ time.

Proof

We recall that

- Only M-processes are checked by the safety algorithm (which may give the $O(mn^2)$ running time at step 2).
- Each M-process at moment t_{n-1} , with the exception of a possible M-process which has allocated resources at the current moment t_n , is converted into an S-process at the current moment t_n .
- An S-process can be converted into an M-process again only if it allocates resources.

It is easy to see that the number of M-processes decreases at each moment and the sum of the number of M-processes at each moment, under q moments, bounded by $q + n$. The bound $q + n$ can be reached if at some moment all the processes are M-processes and if at the $q - 1$ subsequent moments some process allocates resources. It means at most $n(n + 1)/2 + q - 1$ searches under q moments at step 2 in the safety algorithm, that is $n^2/2q + n/2q + 1 - 1/q$ searches per moment and $q \geq cn$ gives $O(mn)$ running time at step 2 in the safety algorithm. It is straightforward to see that the time needed for the other steps of the algorithm is not more than $O(mn)$. $\square$

In the average case:

In the average case, we can only say that the running time is $O(mne)$, where $1 \leq e \leq n$. Our belief is that e is a constant.

The average case depends on several factors such as the number of resource classes, the number of resources in each class and the number of processes. The interplay between these factors is hard to estimate. However, condition C may come up to the average case. It is a reasonable assumption that no process completes its task under $q \geq cn$ moments. If the number of processes in a system (with the same amount of resources) increases then the number of steps until some process completes its task may also increase. It is harder to see that no process is rejected (after checking by the safety algorithm) under $q \geq cn$ moments. However, we have to observe that the bound $n + q$ M-processes, to be checked under q moments, is a worst case bound and the worst case bound in the average case may neutralize the cost of some rejected processes under q moments. Because the amount of M-processes decreases during the period of q moments, the expensive rejections may happen at the beginning of the period. On the other hand, the amount of resources is larger at the begin-

ning of a period implying that the possible rejections may often happen at the end of a period and consequently they may be cheap.

Simulation results (see in the next section !) confirm that the running time may be $O(mn)$ in the average case.

Comparisons:

The banker's algorithm [6,8,18] works in $O(mn^2)$ time both in the worst case and in the average case.

T. Kameda's algorithm [12], employing a network technique, uses the property of a system that not all resource classes always block all the processes. Its running time is $O(mn^{1.5})$ in the worst case (and presumably even in the average case).

The modified banker's algorithm uses information from previous allocation states. Its running time is $O(mn^2)$ in the worst case, $O(mn)$ under condition C and $O(mne)$ in the average case, where $1 \leq e \leq n$ (probably constant).

The modified algorithm can be further improved by "remembering" the processes marked T at the previous moment (which means to apply the technique used in the safety algorithm at step 1). In this way, the bound of the for-loops in the main algorithm will be less than n in the average case.

The simulation, discussed in the next section, shows a drastic improvement. However, we have to pay a little for this improvement by a more restrictive algorithm. It can happen that if the modified banker's algorithm rejects a request of an S-process, with high values in the vector Need, the banker's algorithm will accept it. This situation will not happen often according to the simulation result, and if it happens it might lead to a more efficient resource allocation (compare the number of rejected and completed processes for the two algorithms in the simulation result, given in the next section!).

We should observe that by remarking all of the processes to M in the vector State, the modified algorithm will choose the same safe sequence as the banker's algorithm will (and it will take $O(mn^2)$ time).

3. Simulation

We made a simulation to compare the banker's algorithm, given in [18], and our modified algorithm. All of the start values were chosen as pseudo random numbers and both algorithms were runned by the same series of pseudo random numbers. The processes were served from a circular queue and their requests (between 1 and $\text{Need}[i,j]$, $j = 1, 2, \dots, \text{number of resource classes}$, for each process i) were also chosen by random numbers. The simulation was done for 10, 20, 30, ..., 100 processes where the number of resource classes varied by 5, 10 and 50, and the number of resources of each class varied by 10, 50 and 100. The number of allocations were 10000 for each set of data and the whole simulation was repeated for five different series of random numbers.

For both algorithms, we counted the number of calls of the safety algorithm and the number of searches in the safety algorithm at step 2, i.e., at that part of the algorithm which could give the $O(mn^2)$ running time. We counted also the number of completed processes and the number of rejected requests (at step 2 in the main algorithm or after control by the safety algorithm). All of the result tables showed a drastic improvement in the number of searches for the modified algorithm. The number of completed and rejected processes were the same with 50 resource classes in all result tables while it varied between the two algorithms with 5 and 10 resource classes. However, more processes were completed by the modified algorithm; the relation was 60 - 40 %.

We picked the table in figure 1. from the result tables as a representative table for the simulation. The column, marked 'searches/(calls * n^2)' shows that the banker's algorithm works in $O(mn^2)$ time while the column, marked 'searches/(calls * n)' suggests that the modified banker's algorithm works in $O(mn)$ time (because the values in the columns are almost constant). According to the simulation, the number of searches were extremely low for the modified algorithm in all result tables when the number of resource classes were high (=50). Such a table is given in figure 2.

Assignments	10000											
Resource classes	5											
Limit of resources in each class	50											
banker's algorithm												
modified banker's algorithm												
safety						safety						
proc.	alg. searches <u>searches</u> compl. reject.					alg. searches <u>searches</u> compl. reject.						
n	calls	calls*n ²				calls	calls*n					
10	!	2471	126939	0.51	459	8159	!	1198	1691	0.14	428	8343
20	!	1469	218194	0.37	184	9114	!	1063	3434	0.16	205	9101
30	!	1138	302252	0.30	137	9429	!	960	3631	0.13	153	9420
40	!	1044	381675	0.23	89	9592	!	1080	4714	0.11	102	9587
50	!	723	494633	0.27	73	9644	!	824	6210	0.15	73	9680
60	!	761	570163	0.21	58	9718	!	977	8211	0.14	83	9663
70	!	606	703908	0.24	64	9737	!	600	5169	0.12	63	9754
80	!	558	713560	0.20	42	9797	!	588	6558	0.14	58	9754
90	!	657	1002883	0.19	65	9773	!	534	5444	0.11	55	9813
100	!	519	1121977	0.22	56	9791	!	647	11587	0.18	78	9747

Few resource classes

Figure 1.

Assignments		10000	
Resource classes		50	
Limit of resources in each class		50	

banker's algorithm				modified banker's algorithm			
safety		safety		safety		safety	
proc.	alg. searches	searches compl.	reject.	alg. searches	searches compl.	reject.	
n	calls	calls*n ²		calls	calls*n		
10	!	1149	74685	0.65	165	8851	!
20	!	584	133690	0.57	85	9419	!
30	!	392	191715	0.54	55	9613	!
40	!	300	248980	0.52	40	9711	!
50	!	241	312950	0.52	36	9764	!
60	!	200	367020	0.51	27	9806	!
70	!	171	424480	0.51	23	9834	!
80	!	156	482280	0.48	20	9855	!
90	!	143	541125	0.47	18	9871	!
100	!	122	598000	0.49	16	9884	!

Many resource classes

Figure 2.

Comment to figure 2 :

The question is, why is the number of searches as low for the modified algorithm, and why do the values of the last two columns become same of both algorithms. The only explanation we can give is that with many resource classes the system is sluggish and, either almost all of the processes are T-processes, or the requests are rejected at step 2 in the main algorithm.

4. Conclusion

The banker's algorithm provides a general resource allocation but its running time is $O(mn^2)$. Our modification preserves almost all of the generality of the banker's algorithm while its running time is $O(mn)$ under certain condition, and probably even in the average case according to the simulation result. Furthermore, our modification is a bit restrictive, compared with the banker's algorithm, but it seems to provide a more efficient resource allocation than the banker's algorithm in the long run.

Future computer systems will be oriented more towards asynchronous parallel operation and multiple processor systems will be common. There will, quite simply, be more operations going on concurrently and the number of processes will increase. With the increasing tendency of operating system designers to view data as a resource, the number of resources operating systems must manage might increase dramatically. The extremely low number of searches showed by the simulation result when the number of resource classes were high, suggests that our algorithm will fit well for systems with a large number of processes and resources.

Acknowledgements

The author is grateful to Dr Eric Astor, Svante Carlsson, Dr Rolf Karlsson and Dr Boris Magnusson for their useful suggestions.

References

1. P. A. Bernstein & N. Goodman, "Concurrency Control in Distributed Database Systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.
2. P. A. Bernstein & J. B. Rothnie & N. Goodman & C. A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1 : A System for Distributed Databases", IEEE Trans. on Software Engineering, Vol. SE-4, No. 3, May 1978, pp. 154-168.
3. M. C. Chen & M. Rem, "Deadlock-Freedom in Resource Contentions", Acta Informatica 21, 1985, pp. 585-598.
4. E. G. Coffman & M. J. Elphick & A. Shoshani, "System Deadlocks", Computing Surveys, Vol. 3, No. 2, June 1971, pp. 67-78.
5. H. M. Deitel, "An Introduction to Operating Systems", Addison-Wesley, 1984.
6. E. W. Dijkstra, "Cooperating Sequential Processes", Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.
7. V. D. Gligor & S. H. Shattuck, "On Deadlock Detection in Distributed Systems", IEEE Trans. on Software Engineering, Vol. SE-6, No. 5, September 1980, pp. 435-440.
8. A. N. Haberman, "Prevention of System Deadlocks", Communication of the ACM, Vol. 12, No. 7, July 1969, pp. 373-385.
9. J. W. Havender, "Avoiding Deadlock in Multitasking Systems", IBM System Journal, Vol. 7, No. 2, 1968, pp. 74-84.
10. R. C. Holt, "Some Deadlock Properties of Computer Systems", Computing Surveys, Vol. 4, No. 3, September 1972, pp. 179-196.
11. Toshihide Ibaraki & Tsunehiko Kameda, "Deadlock-Free Systems for Bounded Number of Processes", IEEE Trans. on Computers, Vol. C-31, No. 3, March 1982, pp. 188-193.
12. Tiko Kameda, "Testing Deadlock-Freedom of Computer Systems", Journal of the ACM, Vol. 27, No. 2, April 1980, pp. 270-280.
13. W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.

14. J. Y.-T. Leung & E. K. Lai, "On Minimum Cost Recovery from System Deadlock", IEEE Trans. on Computers, Vol. C-28, No. 9, September 1979, pp. 671-677.
15. D. B. Lomet, "Subsystems of Processes with Deadlock Avoidance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 297-304.
16. H. Madduri & R. Finkel, "Extension of the Banker's Algorithm for Resource Allocation in a Distributed Operating System", Inf. Process. Lett. 19, 1 (July 1984), pp. 1-8.
17. D. A. Menasce & R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 195-202.
18. J. Peterson & A. Silberschatz, "Operating System Concepts", Addison-Wesley, 1983.
19. D. J. Rypka & A. P. Lucido, "Deadlock Detection and Avoidance for Shared Logical Resources", IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, January 1985, pp. 67-80.
20. M. K. Sinha & N. Natarajan, "A Priority Based Distributed Deadlock Detection Algorithm", IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, January 1985, pp. 67-80.
21. M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed I N G R E S", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 188-194.

PART III: A Dynamic Graph Model

DYNAMIC OPERATIONS ON DIRECTED ACYCLIC GRAPHS

Ferenc Belik

Department of Computer Science

Lund University, Sweden

Abstract

The problem of changing a directed acyclic graph while keeping it acyclic involves three operations on edges or nodes, namely deletion, unsuccessful insertion and successful insertion, where an insertion includes a check if a cycle is formed. This problem has an important application in the problem of deadlock avoidance in resource allocation systems, where the unsuccessful insertion of edges is the very most frequent operation. We propose a dynamic technique using a path matrix representation of the graph, which gives $O(1)$ time for detecting a cycle when operating on an edge (v,w) and $O(n \cdot \min(\text{indeg}(q), \text{outdeg}(q)))$ time when operating on a node q . In the case of operations on edges, the unsuccessful insertion operations can amortize the cost of the other two types of operation and linear or even constant amortized time for one operation is realistic in the mentioned application. A comparison between the running time for topological sorting, which is the previously proposed technique for detection of cycles in the mentioned application, and our dynamic technique, when the number of edges is $\theta(n^2)$, shows that our technique is better under certain conditions in the case of operations on nodes and it is superior in the case of operations on edges. It is better with an additive term in deletion and successful insertion while in unsuccessful insertion it is much better. It takes $O(1)$ time while topological sorting takes $\theta(n^2)$.

1. Introduction

We define a directed graph (or digraph) $G = (V, E)$ consisting of a set $V = \{1, 2, \dots, |V|\}$ of nodes and a set $E \subseteq V \times V$ of edges. A pair $(v, w) \in E$ is called an edge from v to w . Throughout this paper we set $n = |V|$ and $e = |E|$. A path from v to w , $v, w \in V$, is a sequence $v_0, v_1, v_2, \dots, v_k$ of nodes such that $v_0 = v$, $v_k = w$ and $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$; k is the length of the path. Note that there is always the path of length zero from v to v . A cycle is a path from v to v . A graph is acyclic if it contains no non-trivial cycle. The indegree of a node v is the number of edges ending in v , $\text{indeg}_G(v) = |\{w ; (w, v) \in E\}|$. Similarly, the outdegree of v is the number of edges starting in v , $\text{outdeg}_G(v) = |\{w ; (v, w) \in E\}|$. To express the running time of the algorithms we also count the number of nodes which can be reached from node v and the number of nodes from which node v can be reached. Let $\text{reach_from}(v) = |\{w_i ; \text{there is a path of length } > 0 \text{ from } v \text{ to } w_i\}|$ and $\text{reach_to}(v) = |\{w_i ; \text{there is a path of length } > 0 \text{ to } v \text{ from } w_i\}|$.

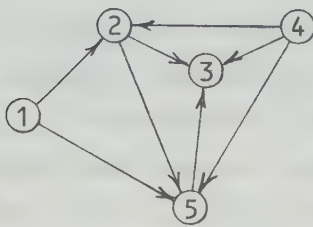
Topological sorting is an efficient algorithm for the detection of cycles in directed graphs [1,2,5-7]. The algorithm uses a graph reduction technique with an adjacency list representation of the graph and it may scan the entire graph to find a cycle. The running time of the algorithm is $O(n + e)$. Suppose that we have an acyclic digraph and we want to change the graph by inserting and deleting edges or nodes while keeping the graph acyclic. This problem demands checking whether the insertion of a new edge or node into the graph makes the graph cyclic. This problem has an important application in the problem of deadlock avoidance in resource allocation systems [3]. It is intuitively clear that it is wasteful to use topological sorting for the detection of a cycles. In this paper we propose a dynamic technique which efficiently solves the problem. A comparison between our technique and the topological sorting is given later.

An appropriate representation of a graph for edgewise detection of cycles and for insertion and deletion of edges is a path matrix, which is a $|V| \times |V|$ matrix $P_G = (p_{ij})_{1 \leq i, j \leq n}$ with

$$p_{ij} = \begin{cases} m & \text{if there are } m \geq 1 \text{ different paths of length } > 0 \\ & \text{from node } i \text{ to node } j \\ 0 & \text{if there is no path of length } > 0 \text{ from node } i \text{ to} \\ & \text{node } j \end{cases}$$

The storage requirement of this representation is clearly $\theta(n^2)$.

Example 1: A graph G and its path matrix



$$P_G = \begin{pmatrix} 0 & 1 & 3 & 0 & 2 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

□

2. The Dynamic Technique

To check if the insertion of an edge (v,w) into a graph G makes the graph cyclic it is sufficient to have a set $P = \{ [v_i, v_j] ; \text{there is a path of length } > 0 \text{ from } v_i \text{ to } v_j, v_i, v_j \in V \}$. If P is represented as a boolean matrix it takes $O(1)$ time to check if $P[w,v] \neq 0$. However, this representation is not enough if we want to be able to delete edges. Instead consider the path matrix representation given in section 1. It is clear that the path matrix is suitable for the detection of cycles in $O(1)$ time. To make the description of the other operations easier we will take a closer look at the path matrix representation.

Theorem 1: Let $G = (V,E)$ be an acyclic digraph and let P_G be its path matrix representation as defined in section 1.

A) Consider node q with edges (r,q) , $r = i,j,\dots,l$ ending in q. Then, column q of the path matrix P_G , denoted by $P_G[_{,}q]$, can be partitioned into $P_G[_{,}i], P_G[_{,}j], \dots, P_G[_{,}l], V_i[_{,}], V_j[_{,}], \dots, V_l[_{,}]$, where

$P_G[_{,r}]$ denotes column r in P_G and $V_r[_{,}]$, $r = i, j, \dots, l$ are column vectors of length n , consisting of zero elements anywhere except at index r which is set to one. The vector $V_r[_{,}]$ corresponds to the immediate path from r to q via the edge (r, q) .

- B) Consider node q with edges (q, r) , $r = i, j, \dots, m$ starting in q . Then, row $P_G[q, _{,}]$ can be partitioned into $P_G[i, _{,}], P_G[j, _{,}], \dots, P_G[m, _{,}]$, $(V_i[_{,}])^T, (V_j[_{,}])^T, \dots, (V_m[_{,}])^T$ where T denotes transpose. The vector $(V_r[_{,}])^T$ corresponds to the "direct" path from q to r via the edge (q, r) .
- C) The path matrix P_G is a unique representation of the graph G .

Proof:

- A) Take an element p_{kq} of column $P_G[_{,}q]$. A value s at p_{kq} means that there are s paths of length > 0 from node k to node q .

If $\text{indeg}_G(q) = 0$ then $s = 0$. Otherwise the theorem says that s can be partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}, V_i[k], V_j[k], \dots, V_l[k]$, where $V_r[k] = 0$ if $r = k$. We have two cases:

- 1) If $s = 0$ then there is no path from k to q , the edge (k, q) cannot exist, which means that $V_k[_{,}]$ does not exist.

Furthermore, for all $r = i, j, \dots, l$, p_{kr} must be zero, that is, there is no path from k to r . Similarly the converse is true, that is, if there is no path from k to q then the partitioning of p_{kq} must consist of zeros.

- 2) There are $s > 0$ paths from k to q . If the edge (k, q) exists then $V_k[k] = 1$ and $s - 1$ is partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}$ since p_{kr} , $r = i, j, \dots, l$, is the number of paths from k to r and there is one path from r to q . On the other hand, if there is no edge (k, q) then $V_k[_{,}]$ does not exist and s is partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}$. The converse is also true, that is, the sum of $p_{ki}, p_{kj}, \dots, p_{kl}$ is s or $s - 1$ depending on whether there exists an edge from k to q .

- B) The proof follows in the same way as in part A.

- C) Since the partitioning is unambiguous the path matrix P_G correctly represents the graph G . $\square$

Example 2: The column $P_G[_{,}3]$ in the path matrix given in Example 1 may be partitioned into $P_G[_{,}2], P_G[_{,}4], P_G[_{,}5], V_2[_{,}], V_4[_{,}], V_5[_{,}]$, that is,

$$P_G[-,3] = \begin{pmatrix} 3 \\ 2 \\ 0 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 0 \\ 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad \square$$

Henceforth, we will use the notations introduced in Theorem 1.

In the following, in context with a path and a node or an edge, we will use the words: starting, ending and passing through. We define the paths starting in a node q as $\{v_0, v_1, \dots, v_k; v_0 = q \text{ and } (v_i, v_{i+1}) \in E\}$, the paths ending in q as $\{v_0, v_1, \dots, v_k; v_k = q \text{ and } (v_i, v_{i+1}) \in E\}$, the paths passing through q as $\{v_0, v_1, \dots, v_k; v_i = q \text{ for some } i \neq 0, k \text{ and } (v_i, v_{i+1}) \in E\}$ and the paths passing through an edge (v, w) as $\{v_0, v_1, \dots, v_k; v_j = v, v_{j+1} = w \text{ where } 0 \leq j < k \text{ and } (v_i, v_{i+1}) \in E\}$. We will use the expression number_of_paths to express the size of these sets. To describe the number of paths passing through a node or an edge we need the definition of the outer product of a column and a row vector. Let V_c be a column vector and V_r a row vector, both of length n . The outer product, denoted by $V_c \otimes V_r$, is a $|V_c| \times |V_r|$ matrix $P = (p_{ij})_{1 \leq i, j \leq n}$ with $p_{ij} = V_c[i] * V_r[j]$.

Theorem 2: Let $G = (V, E)$ be an acyclic digraph and let P_G be its path matrix representation.

- A) The number of paths starting in or passing through a node q is expressed by the path matrix $P_q = (P_G[-, q] + V_q[-]) \otimes P_G[q, -]$.
- B) The number of paths starting in or passing through a node v and passing through the edge (v, w) is expressed by the path matrix $P_{(v, w)} = (P_G[-, v] + V_v[-]) \otimes (P_G[w, -] + (V_w[-])^T)$.

Proof:

- A) The number of paths ending in q is expressed by $P_G[-, q]$ and the starting of paths in q by $V_q[-]$. The proceeding of the paths ending in q corresponds to the paths passing through q . The nodes reachable from q are given by the non-zero elements in row $P_G[q, -]$. The number of paths from q to a node w is given by $P_G[q, w]$. According to Theorem 1, $P_G[q, w] = k$ paths from q to node w means that the paths, represented by columns $P_G[-, q]$ and $V_q[-]$,

enter node w k times. Then, Theorem 1 and the definition of the outer product give the proof.

- B) Note that the paths from v proceed to w and to the nodes corresponding to the non-zero elements of row $P_G[w, -]$. This is expressed by the row vector $(P_G[w, -] + (V_w[-])^T)$ in the outer product. Then, the proof follows in the same manner as in part A. $\square$

Example 3: Consider graph G and its path matrix P_G given in Example 1.

- A) The number of paths starting in or passing through node 2 is expressed by the path matrix $P_2 = (P_G[-, 2] + V_2[-]) \circledast P_G[2, -]$, that is, by

$$P_2 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \circledast (0 \ 0 \ 2 \ 0 \ 1) = \begin{pmatrix} 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- B) The number of paths starting in or passing through node 2 and passing through the edge $(2, 5)$ is expressed by the path matrix

$$P_{(2,5)} = (P_G[-, 2] + V_2[-]) \circledast (P_G[5, -] + (V_5[-])^T), \text{ that is, by}$$

$$P_{(2,5)} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \circledast ((0 \ 0 \ 1 \ 0 \ 0) + (0 \ 0 \ 0 \ 0 \ 1)) = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \square$$

Note that it is not immediate from the path matrix if there exists an edge (v, w) , i.e., the path v, w . To see that, we use an algorithm given in the Appendix.

Deletion of an edge (v,w) means that all the paths starting in or passing through node v and passing through (v,w) must be deleted, which can be expressed by $P_G - P_{(v,w)}$.

Example_4: The deletion of the edge $(2,5)$ from the graph G given in Example 1 is expressed by $P_{G'} = P_G - P_{(2,5)}$:

$$P_{G'} = \begin{pmatrix} 0 & 1 & 3 & 0 & 2 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad \square$$

In the same way, the insertion of an edge (v,w) can be expressed by $P_G + P_{(v,w)}$.

To save time when deleting or inserting an edge (v,w) we add or subtract only the non-zero elements of $P_{(v,w)}$. Let set S contain the non-zero elements of column $(P_G[_ ,v] + V_v[_])$ together with their row indices and let set Q contain the non-zero elements of row $(P_G[w, _] + (V_w[_])^T)$ together with their column indices in the following algorithm:

Algorithm_1: (deletion of an edge (v,w))

```

(1)  S := {};
(2)  for i := 1 to n do
(3)    if  $P_G[i,v] \neq 0$  then
(4)      S := S  $\cup$  {[i,  $P_G[i,v]$ ]};
(5)  S := S  $\cup$  {[v,1]}; (* the non-zero elements of  $(P_G[_ ,v] + V_v[_ ])$  *)
(6)  Q := {};
(7)  for j := 1 to n do
(8)    if  $P_G[w,j] \neq 0$  then
(9)      Q := Q  $\cup$  {[j,  $P_G[w,j]$ ]};
(10) Q := Q  $\cup$  {[w,1]}; (* the non-zero elements of  $(P_G[w, _] + (V_w[_ ])^T)$  *)
(11) for all elements [i,p] in S do
(12)   for all elements [j,t] in Q do
(13)      $P_G[i,j] := P_G[i,j] - p*t$ ; (*  $P_G := P_G - P_{(v,w)}$  *)

```


Theorem_3: Let $G = (V, E)$ be an acyclic graph and let P_G be its path matrix.

- A) Let $G' = (V, E')$, $E' = E - \{(v, w)\}$, be the graph after deleting edge (v, w) from G . Algorithm 1 correctly computes the path matrix $P_{G'}$ of the graph G' .
- B) Let $G'' = (V, E'')$, $(v, w) \notin E$ and $E'' = E \cup \{(v, w)\}$, be the acyclic graph obtained after inserting the edge (v, w) into G . Algorithm 1, with addition instead of subtraction in line (13) correctly computes the path matrix $P_{G''}$ of the graph G'' .
- C) If the element $P[w, v]$ in the path matrix P_G is equal to zero then the insertion of the edge (v, w) into G does not make the graph cyclic.

Proof:

- A) and B): It is easy to see that the algorithm computes $P_G - P_{(v, w)}$. Then, the proof follows from Theorem 1 and Theorem 2.
- C): The proof follows from the correct representation of the graph G by its path matrix P_G according to Theorem 1. $\square$

Theorem_4: Let the sets S and Q in Algorithm 1 be represented as linked lists. Then, the running time of Algorithm 1 is $O(\max(n, \text{reach_to}(v) * \text{reach_from}(w)))$.

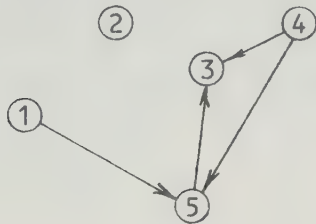
Proof: After line (5) we have $|S| = O(\text{reach_to}(v))$ and after line (10) $|Q| = O(\text{reach_from}(w))$. Then, the running time of lines (11) - (13) is $O(\text{reach_to}(v) * \text{reach_from}(w))$. The running time of lines (2) - (4) and (7) - (9) is $O(n)$ and of lines (1), (5), (6) and (10) $O(1)$. $\square$

Deletion of a node means that all paths ending in, starting in or passing through q must be deleted. The paths ending in q are represented by $P_G[_, q]$ and the paths starting in or passing through q by P_q . Then, the deletion of q can be expressed by setting the elements of $P_G[_, q]$ to zero and subtracting P_q from P_G . This can be done by a simple modification of Algorithm 1 with running_time $O(\max(n, \text{reach_to}(q) * \text{reach_from}(q)))$. Deletion of node q makes column q and row q empty (that is, to consist only of zeros) in the path matrix.

In the case of insertion of a new node into an empty place q (i.e., column q and row q are both empty) we first compute column q and row q . We should observe that the new node obtains the name q corresponding to the empty place. Column q is given by the sum of the columns $P_G[_ , v]$ and $V_v[_]$ corresponding to the nodes v from which there are edges to q . Row q is given by the sum of the rows $P_G[w, _]$ and $(V_w[_])^T$ corresponding to the nodes w to which there are edges from q . Then, after computing P_q , the insertion of node q is expressed by $P_G + P_q$ and by "filling" column $P_G[_ , q]$.

Example 5:

A) Consider graph G and its path matrix P_G given in Example 1 and the path matrix P_2 given in Example 3. The deletion of node 2, expressed by $P_{G'} = P_G - P_2$ and by setting the elements in column 2 to zero, gives the graph G' and its representation:

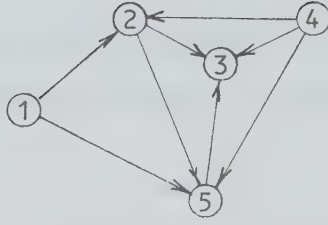


$$P_{G'} = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

B) Consider graph G' and its path matrix $P_{G'}$ in part A). The insertion of a new node 2, represented by the edges $(1,2), (2,3), (2,4), (2,5)$, is expressed by $P_{G'} + P_2$ and by filling column $P_{G'}[_ , 2]$:

$$P_2 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \otimes (0 \ 0 \ 4 \ 1 \ 2) = \begin{pmatrix} 0 & 0 & 4 & 1 & 2 \\ 0 & 0 & 4 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and we obtain



$$P_{G^{1,1}} = \begin{pmatrix} 0 & 1 & 5 & 1 & 3 \\ 0 & 0 & 4 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

□

Let set S contain the computed non-zero elements of column $(P_G[_{,q}] + V_q[_{,}])$ together with their row indices and
 Let set Q contain the computed non-zero elements of row $P_G[q, _]$ together with their column indices in the following algorithm:

Algorithm_2: (insertion of a node q)

```

(1)   $S := \{\}$ ;
(2)  for  $i := 1$  to  $n$  do
(3)    begin
(4)       $p := 0$ ;
(5)      for all  $v; (v, q) \in E$  do
(6)        begin
(7)           $p := p + P_G[i, v]$ ;
(8)          if  $i = v$  then                                (* add  $V_v[_{,}]$  *)
(9)             $p := p + 1$ 
(10)       end;
(11)       if  $p \neq 0$  then
(12)          $S := S \cup \{[i, p]\}$ 
(13)       end;
(14)   $S := S \cup \{[q, 1]\}$   (*  $S$ =non-zero elements of  $P_G[_{,q}] + V_q[_{,}]$  *)
(15)   $Q := \{\}$ ;
(16)  for  $j := 1$  to  $n$  do
(17)    begin
(18)       $t := 0$ ;
(19)      for all  $w; (q, w) \in E$  do
(20)        begin
(21)           $t := t + P_G[w, j]$ ;
(22)          if  $j = w$  then                                (* add  $(V_w[_{,}])^T$  *)
(23)             $t := t + 1$ 
(24)        end;

```

```

(25)   if t <> 0 then
(26)       Q := Q ∪ {[j,t]}
(27)   end;                               (* Q=non-zero elements of PG[q,-] *)
(28)   for all [i,p] in S do
(29)       for all [j,t] in Q do
(30)           PG[i,j] := PG[i,j] + t*p;           (* PG:=PG+Pq *)
(31)   for all [i,p] in S do               (* fill column PG[-,q] *)
(32)       PG[i,q] := p

```

Theorem 5: Let $G = (V, E)$ be an acyclic graph and let P_G be its path matrix.

- A) Let $G' = (V', E')$, $V' = V \cup \{q\}$ where $q \notin V$ and $E' = E \cup \{(v, q), (q, w)\}$ for all $v, w \in V$, be the graph obtained after inserting node q into G . Let the row q and column q be empty in the path matrix P_G . Algorithm 2 then correctly computes the path matrix $P_{G'}$ of the graph G' .
- B) Let (q, w_i) , $i = 1, 2, \dots, l$, be all edges starting in node q . If the set S after line (13) in Algorithm 2 does not contain any elements $[w_i, p]$, $i = 1, 2, \dots, l$, then the insertion of node q does not produce a cycle.
- C) Let (v_i, q) , $i = 1, 2, \dots, m$, be all edges ending in node q . If the set Q after line (27) in Algorithm 2 does not contain any element $[v_i, p]$, $i = 1, 2, \dots, m$, then the insertion of the node q does not produce a cycle.

Proof:

- A) It is easy to see that Algorithm 2 computes $P_{G'} := P_G + P_q$ and fills column $P_{G'}[-, q]$ in $P_{G'}$. Then, the proof follows from the discussion above and from Theorem 1 and Theorem 2.
- B) Set S after line (13) contains all the nodes from which there are paths leading to q via the edges ending in q . Then, S can be used to check whether the edges starting in q makes the graph cyclic.
- C) The proof follows in the same manner as in part B. $\square$

Theorem 5: Let the sets S and Q be represented as linked lists and let the edges ending and starting in node q be represented by adjacency

lists in Algorithm 2.

A) The running time of Algorithm 2 is

$$O(\max(n \cdot \text{indeg}(q), n \cdot \text{outdeg}(q), \text{reach_to}(q) \cdot \text{reach_from}(q))).$$

B) To check if the insertion of the node q makes the graph cyclic takes $O(n \cdot \min(\text{indeg}(q), \text{outdeg}(q)))$ time.

Proof:

A) The running time of lines (2) - (13) is $O(n \cdot \text{indeg}(q))$ and of lines (16) - (27) $O(n \cdot \text{outdeg}(q))$. Since $|S| = \text{reach_to}(q)$ and $|Q| = \text{reach_from}(q)$ at line (28), the running time of lines (28) - (30) is $O(\text{reach_to}(q) \cdot \text{reach_from}(q))$. The running times of the other lines are less; lines (31) - (32) require $O(\text{reach_to}(q))$ and lines (1), (14) and (15) $O(1)$.

B) Let the sets S and Q also be represented by boolean arrays, where element $i = 1$ if $[i, p] \in S$ or $[i, t] \in Q$ respectively.

If $\text{indeg}(q) < \text{outdeg}(q)$ then compute S , which takes $O(n \cdot \text{indeg}(q))$

time. To check for the absence of cycles requires $O(\text{outdeg}(q))$ time.

On the other hand, if $\text{indeg}(q) \geq \text{outdeg}(q)$ then we compute R , which takes $O(n \cdot \text{outdeg}(q))$ time, and the checking for cycles requires

$O(\text{indeg}(q))$ time. $\square$

3. Comparison between the Dynamic Technique and Topological Sorting

Consider the problem mentioned in the section 1, where we have an acyclic digraph $G = (V, E)$ and want to change it, but keep it acyclic. There are three operations involved (both on edges and nodes): 1) deletion, 2) attempted insertion, which does not succeed because it would give a cycle, 3) successful insertion. We can compare our method with topological sorting only in the case when the number of edges in G is maximal, i.e., $e = n(n - 1)/2$, since the running time of our algorithms are not expressed as a function of e . We want to lay stress upon that the running times of our algorithms also decrease when the number of edges decreases. In the topological sorting-algorithm the graph G is represented by adjacency lists while our algorithms represent G by its path matrix P_G .

Operations_on_edges

We have the following running times with respect to an edge (v,w):

	top. sorting	dynamic technique
deletion	$O(\text{outdeg}(v))$	$O(\max(n, \text{reach_to}(v) * \text{reach_from}(w)))$
unsuccessful		
insertion	$O(n + e)$	$O(1)$
successful		
insertion	$O(n + e)$	$O(\max(n, \text{reach_to}(v) * \text{reach_from}(w)))$

When $e = n*(n - 1)/2$, the worst case for our technique occurs when node w succeeds node v in the topological order and both w and v are in the middle of the topological order. Then, $\text{outdeg}(v) = \lfloor n/2 \rfloor$, $\max(n, \text{reach_to}(v) * \text{reach_from}(w)) = \lfloor n/2 \rfloor * \lfloor (n - 1)/2 \rfloor$ and $n + e = n^2/2 + n/2$. Since the number of deletions and the number of successful insertions must be almost the same if the number of these operations are large, we can compare one deletion and one successful insertion together for both techniques. We find that $n + e + \text{outdeg}(v) > 2 * \max(n, \text{reach_to}(v) * \text{reach_from}(w))$, meaning that one deletion and one successful insertion together takes less time with our method. Then, our technique with its $O(1)$ running time for an unsuccessful insertion is superior to topological sorting which, has a running time of $O(n + e)$. The cost of the unsuccessful insertion operation can amortize the cost of the other two types of operations (over a sequence of operations with no real time constraint). If for each deletion and successful insertion there are n unsuccessful insertions then the cost of one operation is linear, and if there are n^2 unsuccessful insertions then the cost is constant.

Operations_on_nodes

We first give the algorithm for unsuccessful and successful insertions of a node q, using the notations from Algorithm 2 in the previous section.

Algorithm 3:

```
1) if indeg(q) < outdeg(q)
    then begin
        compute S;
        if there is a cycle
            then Exit (* unsuccessful insertion *)
            else compute Q
        end
    else begin
        compute Q;
        if there is a cycle
            then Exit (* unsuccessful insertion *)
            else compute S
        end
2) Compute  $P_G := P_G + P_q$  and (* successful insertion *)
    fill column  $P_G[_ , q]$ .
```

In this case we do not obtain as good a running time as for operations on edges, since the checking for cycles requires initial operations on edges ending or starting in q. On the other hand, topological sorting takes the same time for operations on nodes as for operations on edges (even less time for deletion). We have the following running times on a node q:

	top. sorting	dynamic technique
deletion	$O(1)$	$O(\max(n, \text{reach_to}(q) * \text{reach_from}(q)))$
unsuccessful		
insertion	$O(n + e)$	$O(n * \min(\text{indeg}(q), \text{outdeg}(q)))$
successful		
insertion	$O(n + e)$	$O(\max(n * \text{indeg}(q), n * \text{outdeg}(q), \text{reach_to}(q) * \text{reach_from}(q)))$

When $e = n(n - 1)/2$, then the worst case with our technique occurs for successful insertion when node q is the first or the last node in the topological order, and for deletion and for unsuccessful insertion when node q is in the middle of the topological order. Then, $n + e = n^2/2 + n/2$, and

$$\max(n \cdot \text{indeg}(q), n \cdot \text{outdeg}(q), \text{reach_to}(q) \cdot \text{reach_from}(q)) = n(n - 1),$$

$$\max(n, \text{reach_to}(q) \cdot \text{reach_from}(q)) = \lceil (n - 1)/2 \rceil * \lfloor (n - 1)/2 \rfloor \text{ and}$$

$$n \cdot \min(\text{indeg}(q), \text{outdeg}(q)) = n \cdot \lfloor (n - 1)/2 \rfloor.$$

We can compare the running times of deletion and a successful insertion together in the two algorithms as in the case of operations on edges. Subtracting the costs we obtain $((n - 1)^2/4 + n(n - 1)) - (n^2/2 + n/2) \approx 0.75n^2 - 2n$. It means that our algorithm takes about $0.75n^2 - 2n$ more time for the two operations. On the other hand, our technique gives a better running time for unsuccessful insertions. If the number of unsuccessful insertions is much higher than the number of deletions and successful insertions, which is true for the application [4] mentioned in section 1, then the unsuccessful insertions can amortize the excess cost for the other two operations. Topological sorting takes an additive term n more time for unsuccessful insertions. Thus, if for each deletion and successful insertion we have $0.75n$ unsuccessful insertions then the two algorithms have approximately the same running time. If the number of unsuccessful insertions is less than $0.75n$ then our algorithm is worse than topological sorting; if it is more than $1.5n$ then our algorithm is better.

4. Conclusion

The problem of changing an acyclic digraph while keeping it acyclic has an important application in the problem of deadlock avoidance in resource allocation systems. Of the three operations in the problem, unsuccessful insertion of edges is most frequent in this application according to simulations presented in [4]. Our technique, using a path matrix representation of an acyclic digraph, gives efficient algorithms for the unsuccessful insertion operations. A comparison between our technique and topological sorting when $e = \Theta(n^2)$ shows that our technique is superior in the case of operations on edges. Our technique is

better by an additive term in deletion and successful insertion while in unsuccessful insertion our technique is much better. It gives $O(1)$ running time while topological sorting gives $\Theta(n + e)$. The unsuccessful insertion operation can amortize the cost of the other two types of operation. If for each deletion and successful insertion there are n unsuccessful insertions then the amortized cost of one operation is linear, and if there are n^2 unsuccessful insertions then the amortized cost of an operation is constant. By amortized cost we mean the time of an operation averaged over a worst-case sequence of operations. This latter situation may happen in overloaded resource allocation systems. In the case of operations on nodes, our technique is better when for each deletion and successful insertion there are more than $0.75 \cdot n$ unsuccessful insertions, which is a realistic assumption for the mentioned application.

Acknowledgements

The author is grateful to Svante Carlsson for his proposal to the representation of the paths passing through a node or an edge as an outer product and to Dr Rolf Karlsson for his useful suggestions.

Appendix

Algorithm for determining from the path matrix whether an edge (v, w) exists

Let $G = (V, E)$ be an acyclic digraph which is represented by its path matrix P_G . The number of paths from v to w is given by element $P_G[v, w]$ in P_G . To determine whether the edge (v, w) (i.e., the path v, w) exists we want to obtain the number of paths from v to w passing through other nodes. The nodes from which there are paths to w are given by the non-zero elements in column w and the nodes to which there are paths from

v are given by the non-zero elements in row v in P_G . Let the indices of these non-zero elements in column w and row v form the sets $S_{to(w)}$ and $S_{from(v)}$ respectively. Then, $S = S_{from(v)} \cap S_{to(w)}$ gives the nodes to which there are paths from v and from which there are paths to w . The sum of the elements in row $P_G[v, _]$ corresponding to indices in S gives the number of paths from v to w passing through other nodes. If this sum is equal to $P_G[v, w]$ then the edge (v, w) does not exist, and if the sum is equal to $P_G[v, w] - 1$ then (v, w) exists.

Example 6: Consider graph G and its path matrix P_G given in Example 1. We determine whether the edge $(4, 3)$ exists in the following way:

- (1) $S_{from(4)} = \{2, 3, 5\}$ and $S_{to(3)} = \{1, 2, 4, 5\}$
- (2) $S = \{2, 5\}$
- (3) $P_G[4, 2] + P_G[4, 5] = 3 = P_G[4, 3] - 1$,
that is, the edge $(4, 3)$ exists. □

Algorithm 4:

- (1) $S_{from(v)} := \{\}$;
- (2) $S_{to(w)} := \{\}$;
- (3) for $j := 1$ to n do
- (4) begin
- (5) if $P_G[v, j] \neq 0$ then
- (6) $S_{from(v)} := S_{from(v)} \cup \{j\}$;
- (7) if $P_G[j, w] \neq 0$ then
- (8) $S_{to(w)} := S_{to(w)} \cup \{j\}$
- (9) end;
- (10) $S := S_{from(v)} \cap S_{to(w)}$;
- (11) $p := 0$;
- (12) for all j in S do
- (13) $p := p + P_G[v, j]$;
- (14) if $p = (P_G[v, w] - 1)$ then $(v, w) \in E$;
- (15) if $p = P_G[v, w]$ then $(v, w) \notin E$

Theorem 6: Let $G = (V, E)$ be an acyclic digraph which is represented by its path matrix P_G .

- A) Algorithm 4 correctly determine whether an edge $(v, w) \in E$.
- B) The running time of Algorithm 4 is $O(n)$.

Proof:

- A) The proof follows from the discussion above and from Theorem 1.
 B) It is clear that the algorithm with the sets represented as boolean arrays runs in $O(n)$ time. $\square$

References

1. A. V. Aho & J. E. Hopcroft & J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1975.
2. A. V. Aho & J. E. Hopcroft & J. D. Ullman, "Data Structures and Algorithms", Addison-Wesley, 1983.
3. F. Belik, "An Efficient Deadlock Avoidance Technique", Unpublished.
4. F. Belik, "Deadlock Avoidance with a Modified Bankers Algorithm", LUNFD 6/(NFCS-3008), Dept. of Computer Science, Lund University, Sweden, Sept. 1985.
5. D. E. Knuth, "The Art of Computer Programming, Volume 1" Addison-Wesley, 1975.
6. K. Mehlhorn, "Graph Algorithms and NP-Completeness", Springer Verlag, 1984.
7. R. E. Tarjan, "Data Structures and Networks Algorithms", SIAM Monograph, 1983.

AN EFFICIENT DEADLOCK AVOIDANCE TECHNIQUE

Ferenc Belik

Department of Computer Science,
Lund University, Sweden

Abstract

We present a dynamic technique which is considerable better for detecting deadlocks, when allocating a resource in systems with single instances of each resource type, than topological sorting, the previously proposed technique. We consider the deadlock avoidance problem as the problem of changing a directed acyclic graph while keeping it acyclic. The resource allocation algorithms then involve three operations on edges or on nodes: deletion, unsuccessful insertion and successful insertion where the insertions include a previous detection of cycles. In resource allocation systems rejected requests, that is, unsuccessful insertion of edges, are the most frequent operations. Our dynamic technique, using a path matrix representation of the resource allocation graph, detects a cycle (i.e., an unsuccessful insertion) in $O(1)$ time when operating on edges, so our technique uses only $O(1)$ time to detect deadlock when the system allocates a constant number of resources to a process. This $O(1)$ cost of the unsuccessful insertion operation can amortize the cost of the other two types of operation and linear (or even constant) amortized time for one operation is realistic in resource allocation systems. A comparison with topological sorting, when the number of edges is $\theta(n^2)$, shows that our technique is superior. It is better by an additive term when allocating a resource and it is much better when rejecting a resource request, requiring only $O(1)$ time while topological sorting needs $\theta(n^2)$.

1. Introduction

A situation called deadlock was pointed out by Dijkstra [10] in a computer system in which two or more processes coexist sharing resources. A simple example is where a process P_1 holds a device A and requires device B to continue, but device B is already held by another process P_2 which require A to continue. There is a circular wait here and it will not be resolved unless a "drastic" action, i.e., preemption is taken. Informally, a deadlock is a situation in which there is a process whose progress is blocked forever owing to lack of resources, regardless of the future activity (e.g., release of resource unit) of unblocked processes. Different aspects of the deadlock problem have been investigated in [3,8,12-16,19,25], and new interest in it has been revived in the database and distributed environment [5,6,11,18,20,21,23,26,27]. Two necessary and sufficient conditions for the absence of deadlock were established under the expedient allocation policy in [7]. Coffman et al. [8] classify approaches dealing with deadlocks into three categories: detection and recovery, avoidance and prevention. This paper considers a deadlock avoidance algorithm in systems with single instances of each resource type. The resource allocation states in such systems can be described by resource allocation graphs with the deadlock situations expressed by cycles in the graphs. Topological sorting is an efficient algorithm used for the detection of cycles in directed graphs [1,2,17,22,28] and it is the previously proposed technique for detection of deadlocks. The algorithm uses a graph reduction technique with an adjacency list representation of the graph and it may scan the whole graph to find a cycle. The running time of the algorithm is $O(n + e)$, where n is the number of nodes and e is the number of edges in the graph. However, it is wasteful to scan the whole graph as topological sorting does. In resource allocation systems only a few resources (in most cases only one resource) are allocated or released to or from a process at a time. It means that the resource allocation graph is changed by insertion or deletion of a few edges. Further, the graph may also be changed by insertion or deletion of nodes (that is, processes or resources). The resource allocation problem can then be considered as the problem of changing an acyclic digraph while keep-

ing it acyclic by involving three operations on edges or nodes. These operations are: deletion, unsuccessful insertion and successful insertion where the insertions include a previous detection of cycles. In this paper we propose a dynamic technique which efficiently solves the problem. In resource allocation systems rejected requests, that is, unsuccessful insertions of edges, are the most frequent operations [3]. Our dynamic technique, using a path matrix representation of the resource allocation graph, gives $O(1)$ time for detection of cycles (i.e., for unsuccessful insertions) when operating on edges. That is, our technique uses only $O(1)$ time for detection of deadlocks when the system allocates a constant number of resources to a process. Thus, this $O(1)$ cost can amortize the cost of the other two types of operation and linear (or even constant) amortized time for one operation is realistic in resource allocation systems. By amortized time we mean the time of an operation averaged over a worst-case sequence of operations. A comparison between our technique and topological sorting is given in [4].

In recent computer systems [9,24] deadlock has generally been viewed as a limited annoyance. Most systems implement the basic deadlock prevention methods suggested by Havender [13], and these methods seem to be satisfactory. In future systems, however, resource allocation will tend to be more dynamic and deadlock be a far more critical consideration.

2. Definitions and Representation of an Acyclic Digraph

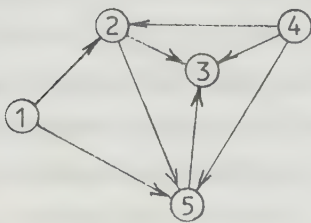
We define a directed graph (or digraph) $G = (V, E)$ consisting of a set $V = \{1, 2, \dots, |V|\}$ of nodes and a set $E \subseteq V \times V$ of edges. A pair $(v, w) \in E$ is called an edge from v to w . A path from v to w , $v, w \in V$, is a sequence $v_0, v_1, v_2, \dots, v_k$ of nodes such that $v_0 = v$, $v_k = w$ and $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$; k is the length of the path. Note that there is always the path of length zero from v to v . A cycle is a path from v to v . A graph is acyclic if it contains no non-trivial cycles. The indegree of a node v is the number of edges ending in v , $\text{indeg}_G(v) = |\{w ; (w, v) \in E\}|$. Similarly, the outdegree of v is the number of edges starting in v , $\text{outdeg}_G(v) = |\{w ; (v, w) \in E\}|$.

An appropriate representation of a graph for edgewise detection of cycles and for insertion and deletion of edges is a path_matrix, which is a $|V| \times |V|$ matrix $P_G = (p_{ij})_{1 \leq i, j \leq n}$ with

$$p_{ij} = \begin{cases} m & \text{if there are } m \geq 1 \text{ different paths of length } > 0 \\ & \text{from node } i \text{ to node } j \\ 0 & \text{if there is no path of length } > 0 \text{ from node } i \text{ to} \\ & \text{node } j \end{cases}$$

The storage requirement of this representation is clearly $\theta(n^2)$.

Example_1: A graph G and its path matrix



$$P_G = \begin{bmatrix} 0 & 1 & 3 & 0 & 2 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

□

To check if the insertion of an edge (v, w) into a graph G makes the graph cyclic it is sufficient to have a set $P = \{ [v_i, v_j] \}$; there is a path of length > 0 from v_i to v_j , $v_i, v_j \in V$. If P is represented as a boolean matrix it takes $O(1)$ time to check if $P[w, v] \neq 0$. However, this representation is not enough if we want to be able to delete edges. Instead consider the path matrix representation given above. It is clear that the path matrix is suitable for the detection of cycles in $O(1)$ time. To make the description of the other operations easier we will take a closer look at the path matrix representation.

Theorem_1: Let $G = (V, E)$ be an acyclic digraph and let P_G be its path matrix representation as defined above.

A) Consider node q with edges (r, q) , $r = i, j, \dots, l$ ending in q. Then, column q of the path matrix P_G , denoted by $P_G[-, q]$, can be parti-

- tioned into $P_G[-,i], P_G[-,j], \dots, P_G[-,l], V_i[-], V_j[-], \dots, V_l[-]$, where $P_G[-,r]$ denotes column r in P_G and $V_r[-]$, $r = i, j, \dots, l$ are column vectors of length n , consisting of zero elements anywhere except at index r which is set to one. The vector $V_r[-]$ corresponds to the immediate path from r to q via the edge (r, q) .
- B) Consider node q with edges (q, r) , $r = i, j, \dots, m$ starting in q . Then, row $P_G[q, -]$ can be partitioned into $P_G[i, -], P_G[j, -], \dots, P_G[m, -], (V_i[-])^T, (V_j[-])^T, \dots, (V_m[-])^T$ where T denotes transpose. The vector $(V_r[-])^T$ corresponds to the immediate path from q to r via the edge (q, r) .
- C) The path matrix P_G is a unique representation of the graph G .

Proof:

- A) Take an element p_{kq} of column $P_G[-,q]$. A value s at p_{kq} means that there are s paths of length > 0 from node k to node q .
- If $\text{indeg}_G(q) = 0$ then $s = 0$. Otherwise the theorem says that s can be partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}, V_i[k], V_j[k], \dots, V_l[k]$, where $V_r[k] = 0$ if $r = k$. We have two cases:
- 1) If $s = 0$ then there is no path from k to q , the edge (k, q) cannot exist, which means that $V_k[-]$ does not exist. Furthermore, for all $r = i, j, \dots, l$, p_{kr} must be zero, that is, there is no path from k to r . Similarly the converse is true, that is, if there is no path from k to q then the partitioning of p_{kq} must consist of zeros.
 - 2) There are $s > 0$ paths from k to q . If the edge (k, q) exists then $V_k[k] = 1$ and $s - 1$ is partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}$ since p_{kr} , $r = i, j, \dots, l$, is the number of paths from k to r and there is one path from r to q . On the other hand, if there is no edge (k, q) then $V_k[-]$ does not exist and s is partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}$. The converse is also true, that is, the sum of $p_{ki}, p_{kj}, \dots, p_{kl}$ is s or $s - 1$ depending on whether there exists an edge from k to q .
- B) The proof follows in the same way as in part A.
- C) Since the partitioning is unambiguous the path matrix P_G correctly represents the graph G . $\square$

Example 2: The column $P_G[-,3]$ in the path matrix given in Example 1 may be partitioned into $P_G[-,2], P_G[-,4], P_G[-,5], V_2[-], V_4[-], V_5[-]$, that is,

$$P_{G[-,3]} = \begin{pmatrix} 3 \\ 2 \\ 0 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 0 \\ 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad \square$$

Henceforth, we will use the notations introduced in Theorem 1, and in context with a path and a node or an edge, we will use the words: starting, ending and passing through. We define the paths starting in a node q as $\{v_0, v_1, \dots, v_k; v_0 = q \text{ and } (v_i, v_{i+1}) \in E\}$, the paths ending in q as $\{v_0, v_1, \dots, v_k; v_k = q \text{ and } (v_i, v_{i+1}) \in E\}$, the paths passing through q as $\{v_0, v_1, \dots, v_k; v_i = q \text{ for some } i \neq 0, k \text{ and } (v_i, v_{i+1}) \in E\}$ and the paths passing through an edge (v, w) as $\{v_0, v_1, \dots, v_k; v_j = v, v_{j+1} = w \text{ where } 0 \leq j < k \text{ and } (v_i, v_{i+1}) \in E\}$. We will use the expression number of paths to express the size of these sets. To describe the number of paths passing through a node or an edge we need the definition of the outer product of a column and a row vector. Let V_c be a column vector and V_r a row vector, both of length n . The outer product, denoted by $V_c \otimes V_r$, is a $|V_c| \times |V_r|$ matrix $P = (p_{i,j})_{1 \leq i, j \leq n}$ with $p_{ij} = V_c[i] * V_r[j]$.

Theorem 2: Let $G = (V, E)$ be an acyclic digraph and P_G its path matrix representation.

- A) The number of paths starting in or passing through a node q is expressed by the path matrix $P_q = (P_{G[-,q]} + V_q[-]) \otimes P_{G[q,-]}$.
- B) The number of paths starting in or passing through a node v and passing through the edge (v, w) is expressed by the path matrix $P_{(v,w)} = (P_{G[-,v]} + V_v[-]) \otimes (P_{G[w,-]} + (V_w[-])^T)$.

Proof:

- A) The number of paths ending in q is expressed by $P_{G[-,q]}$ and the starting of paths in q by $V_q[-]$. The proceeding of the paths ending in q corresponds to the paths passing through q . The nodes reachable from q are given by the non-zero elements in row $P_{G[q,-]}$. The number of paths from q to a node w is given by $P_{G[q,w]}$. According to Theorem 1, $P_{G[q,w]} = k$ paths from q to node

w means that the paths, represented by columns $P_G[-,q]$ and $V_q[-]$, enter node w k times. Then, Theorem 1 and the definition of the outer product give the proof.

- B) Note that the paths from v proceed to w and to the nodes corresponding to the non-zero elements of row $P_G[w,-]$. This is expressed by the row vector $(P_G[w,-] + (V_w[-])^T)$ in the outer product. The proof then follows in the same manner as in part A). $\square$

Example 3: Consider graph G and its path matrix P_G given in Example 1.

- A) The number of paths starting in or passing through node 2 is expressed by the path matrix $P_2 = (P_G[-,2] + V_2[-]) \circledast P_G[2,-]$, that is, by

$$P_2 = \left(\begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \right) \circledast (0 \ 0 \ 2 \ 0 \ 1) = \begin{pmatrix} 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \square$$

Note that it is not immediate from the path matrix if there exist an edge (v,w) (i.e., a path v,w). To see that, we have to use an algorithm with a running time of $O(n)$ [4].

3. Deadlock Avoidance

In order to avoid deadlock in a system, we need to have some additional information on how each process will request resources. There are various models that differ in the amount of such information provided. The simplest and most useful model requires that each pro-

cess declare all the resources that it may need. Given such a priori information it is possible to construct an algorithm that will ensure the system never to enter a deadlock state. This algorithm defines the deadlock avoidance approach.

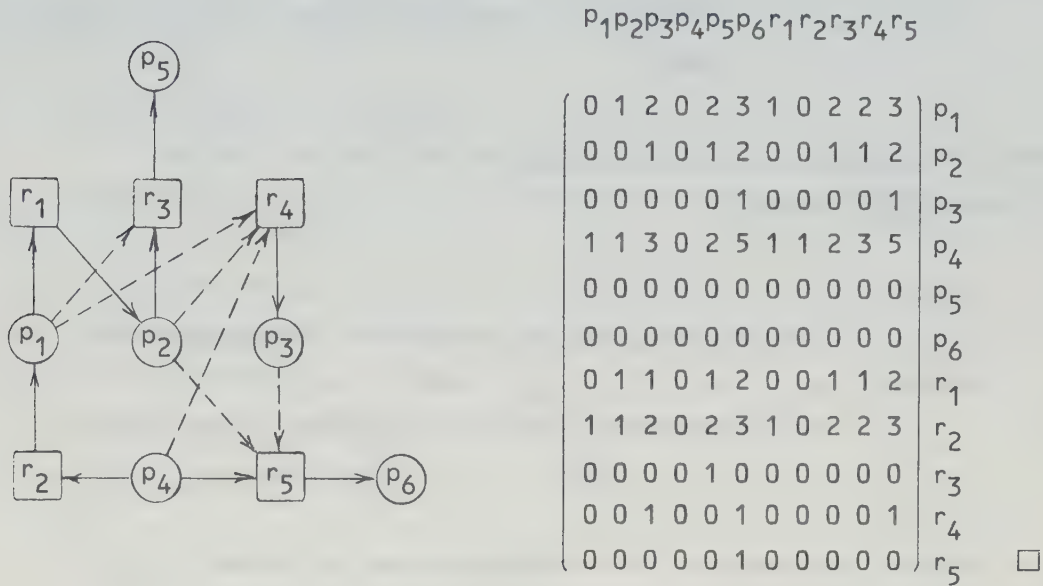
3.1 The Resource Allocation Graph and its Representation

Deadlocks can be described more precisely in terms of a directed bipartite graph called a system resource allocation graph. This graph consists of a pair $G = (V, E)$ where V is a set of vertices and E is a set of edges. The set of vertices is partitioned into two types $P = \{p_1, p_2, \dots, p_n\}$, the set consisting of all processes in the system, and $R = \{r_1, r_2, \dots, r_m\}$, the set consisting of all resources in the system.

Each element in the set E of edges is a ordered pair (p_i, r_j) or (r_j, p_i) , where p_i is a process ($p_i \in P$) and r_j is a resource ($r_j \in R$). If $(p_i, r_j) \in E$, then there is a directed edge from process p_i to resource r_j implying that process p_i requested resource r_j and is currently waiting for that resource. If $(r_j, p_i) \in E$, then there is a directed edge from resource r_j to process p_i implying that resource r_j has been allocated to process p_i . An edge (p_i, r_j) is called a request edge, while an edge (r_j, p_i) is called an assignment edge. In addition to the request and assignment edges, we have a third type of edge called a claim edge. A claim edge $\langle p_i, r_j \rangle$ indicates that process p_i may request resource r_j sometime in the future. This edge resembles a request edge in direction, but is represented by a dashed line in figures. When process p_i requests resource r_j , the claim edge $\langle p_i, r_j \rangle$ is converted to a request edge. When resource r_j is allocated to process p_i the request edge (p_i, r_j) is converted to an assignment edge (r_j, p_i) , and similarly, when a resource r_j is released by p_i , the assignment edge (r_j, p_i) is reconverted to a claim edge $\langle p_i, r_j \rangle$ if the process may request the resource again in the future, otherwise the edge can be deleted. We note that the resources must be claimed a priori in the system. That is, before process p_i starts executing, all of its claim edges must already appear in the resource allocation graph.

Pictorially, we will represent each process p_i as a circle and each resource r_j as a square.

Example 4: A resource allocation graph G and its path matrix:



We can save both time and storage by improving the representation given in Section 2 and used in Example 4 above. The row p_i in the path matrix gives the number of paths from the node corresponding to process p_i to all nodes in the resource allocation graph. Note that there exists only one edge (r_j, p_i) for each resource $r_j, j = 1, 2, \dots, m$, and no edges between processes or between resources.

Then, there are almost the same number of paths from r_j to the other nodes as from p_i , that is, the rows r_j and p_i are almost equal. The only difference is that element p_i in row p_i is zero while in row r_j it is one, corresponding to the path r_j, p_i . Thus, it is sufficient to store the rows $p_1, p_2, \dots, p_n$, and in a vector R_G store the edges $(r_j, p_i), j = 1, 2, \dots, m$. Furthermore, the path matrix can be further reduced by storing only the columns $r_1, r_2, \dots, r_m$. This representation would correctly represent the resource allocation graph since the number of paths leading to p_i via the edges (r_j, p_i) are given by the sum of the columns $P_G[-, r_j]$, where $R_G[r_j] = i$. In this way we obtain the condensed representation of the resource allocation

graph consisting of a path matrix, which is an $n \times m$ matrix

$$P_G = (p_{p_i r_j})_{1 \leq i \leq n, 1 \leq j \leq m} \text{ with}$$

$$p_{p_i r_j} = \begin{cases} k & \text{if there are } k \geq 1 \text{ different paths of length } > 0 \\ & \text{from node } p_i \text{ to node } r_j \\ 0 & \text{if there is no path of length } > 0 \text{ from node } p_i \text{ to} \\ & \text{node } r_j \end{cases}$$

and of a resource allocation vector, which is a m - vector

$$R_G = (r_j)_{1 \leq j \leq m} \text{ with}$$

$$r_j = \begin{cases} i & \text{if resource } j \text{ is allocated to process } p_i \\ 0 & \text{if resource } j \text{ is not allocated to any process} \end{cases}$$

The storage requirement of this representation is $\theta(n \cdot m)$.

Example_5: The condensed representation of the graph G given in Example 4 is:

$$P_G = \begin{matrix} & r_1 r_2 r_3 r_4 r_5 \\ \begin{pmatrix} 1 & 0 & 2 & 2 & 3 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 2 & 3 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix} \end{matrix} \quad R_G = \begin{matrix} & p_i \\ \begin{pmatrix} 2 \\ 1 \\ 5 \\ 3 \\ 6 \end{pmatrix} & \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix} \end{matrix}$$

□

The results of Theorem 1 and Theorem 2 are also valid for the condensed representation of the resource allocation graph.

Theorem 3: Let $G = (V, E)$ be a resource allocation graph represented by its path matrix P_G and resource allocation vector R_G as defined above.

- A) 1. Consider node r_j with edges (p_i, r_j) , $i = a, b, \dots, f$, ending in r_j . Then, column $P_G[-, r_j]$ of the path matrix P_G can be partitioned into

$$R_G[r_k]=a \sum P_G[-, r_k], R_G[r_k]=b \sum P_G[-, r_k], \dots, R_G[r_k]=f \sum P_G[-, r_k],$$

$$V_{p_a}[-], V_{p_b}[-], \dots, V_{p_f}[-]$$

where columns $P_G[-, r_k]$ are different and $V_{p_i}[-]$, $i = a, b, \dots, f$,

are column vectors of length n consisting of zero elements everywhere except at index p_i which is set to one.

2. Consider node p_i with edges (p_i, r_j) , $j = a, b, \dots, g$, starting in p_i . Then, row $P_G[p_i, -]$ can be partitioned into

$$P_G[p_{R_G[r_a]}, -], P_G[p_{R_G[r_b]}, -], \dots, P_G[p_{R_G[r_g]}, -],$$

$$(V_{r_a}[-])^T, (V_{r_b}[-])^T, \dots, (V_{r_g}[-])^T$$

where $V_{r_j}[-]$ are column vectors of length m consisting of zero

elements everywhere except at index r_j which is set to one.

- B) The path matrix P_G and the resource allocation vector R_G gives an unique representation of the graph G .

Proof:

- A) Note in case 1 that the columns $P_G[-, r_k]$, $R_G[r_k] = a, b, \dots, f$, are different since a resource can only be allocated to one process. Furthermore, in case 2, the non-condensed representation of the resource allocation graph, $P_G[p_i, -]$ can be partitioned into $P_G[r_a, -], P_G[r_b, -], \dots, P_G[r_g, -]$,

$$(V_{r_a}[-])^T, (V_{r_b}[-])^T, \dots, (V_{r_g}[-])^T$$

according to Theorem 1; and a row $P_G[r_j, -]$ in the non-condensed representation is equal to $P_G[p_{R_G[r_j]}, -] + (V_{p_{R_G[r_j]}}[-])^T$ in

the condensed representation according to the discussion above,

where $V_{p_{R_G[r_j]}}[-]$ and $V_{r_j}[-]$ are column vectors of length $m+n$ consisting of zero elements everywhere except at index $p_{R_G[r_j]}$ and r_j respectively, which are set to one. However, the partitioning does not contain the vectors $V_{p_{R_G[r_j]}}[-]$ since the condensed representation does not contain the columns $p_{R_G[r_j]}$, $j = a, b, \dots, g$.

The rest of the proof follows from the discussion above and from Theorem 1.

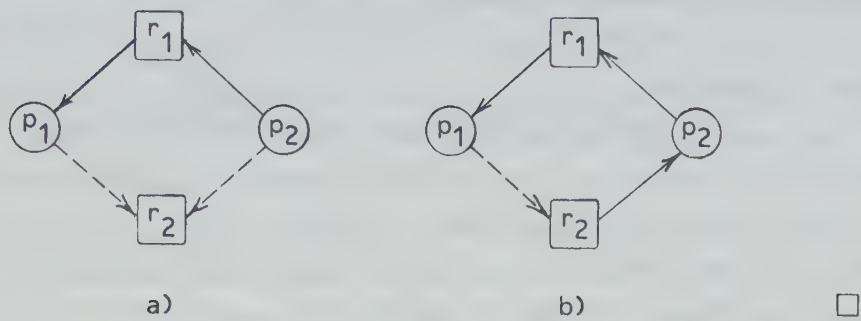
B) The proof follows from part A) and Theorem 1. $\square$

Henceforth, we will also use the notations from Theorem 3.

3.2 The Dynamic Technique Used in the Resource Allocation Algorithm

Suppose that process p_i requests resource r_j . The request can be granted only if reversing the request edge $\langle p_i, r_j \rangle$ to an assignment edge $\langle r_j, p_i \rangle$ does not result in a cycle in the resource allocation graph. The safety check is done by a cycle detection algorithm. If no cycle exists, the allocation of the resource will leave the system in a safe state. Conversely if a cycle is found, the process p_i have to wait for its request to be satisfied, since the allocation would put the system in an unsafe state. To illustrate the algorithm, consider the resource allocation graph of Example 6 a) and suppose that p_2 requests r_2 . Although r_2 is currently free, we cannot allocate it to p_2 since this would create a cycle in the graph (Example 6 b)). A cycle indicates that the system is in an unsafe state since if p_1 then requests r_2 , a deadlock will occur.

Example 6: A safe and an unsafe state in a simple system:



More formally, we define a resource allocation system to be in an unsafe state if the resource allocation graph contains a cycle. If there is no cycle in the resource allocation graph then we say the system is in a safe state.

The resource allocation algorithm can be considered as an algorithm which changes the resource allocation graph while keeping it acyclic. There are three operations in the algorithm on edges or on nodes: deletion, unsuccessful insertion, and successful insertion, where the insertions include a previous detection of cycles. Operations on edges correspond to resource allocation while operations on nodes correspond to insertion (or deletion) of processes or resources to (from) the system. An unsuccessful insertion means that the requesting process has to wait for its request(s) to be satisfied. In the following, we do not differentiate between a claim edge and a request edge, that is, we assume that a process needs all its claimed resources to finish its task.

Our representation of a resource allocation graph makes it possible to detect in constant time whether the insertion of an edge (r_v, p_w) leads to an unsafe state of the system.

Theorem 4: Let a resource allocation system be described by a resource allocation graph $G = (V, E)$, represented by a path matrix P_G and a resource allocation vector R_G . Then the allocation of resource

r_v to process p_w , that is, the insertion of the edge (r_v, p_w) into G , may put the system in an unsafe state if and only if $P_G[p_w, r_v] > 1$. The cost of this operation is $O(1)$.

Proof: We assume that $R_G[r_v] = 0$, that is, resource r_v is not allocated to any process, it can otherwise be checked in $O(1)$ time.

We first prove correctness. It is clear that the insertion of an edge (j, i) into an acyclic digraph makes the graph cyclic if and only if there exists a path from i to j . In our resource allocation system a resource is always claimed or requested in advance, that is, there always exist a path p_w, r_v in G corresponding to the edge (p_w, r_v) when process p_w requests resource r_v . Therefore, the insertion of the edge (r_v, p_w) puts the system in an unsafe state if and only if $P_G[p_w, r_v] > 1$. It is clear that the checking whether $P_G[p_w, r_v] > 1$ takes $O(1)$ time. $\square$

The paths passing through a node or an edge in the resource allocation graph can also be expressed as an outer product.

Theorem 5: Let $G = (V, E)$ be a resource allocation graph and let it be represented by the path matrix P_G and the resource allocation vector R_G .

- A) 1. The number of paths starting in or passing through a node p_q is expressed by the path matrix

$$P_{p_q} = (V_{p_q}[-] + \sum_{R_G[r_i]=q} P_G[-, r_i]) \otimes P_G[p_q, -], \text{ and by}$$

$$R_G[r_j] = w_j \text{ for } P_G[p_q, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq w_j \leq n.$$

2. The number of paths starting in or passing through a node r_q is expressed by the path matrix $P_{r_q} = P_G[-, r_q] \otimes P_G[p_{R_G[r_q]}, -]$,

$$\text{and by } R_G[r_j] = w_j \text{ for } P_G[p_{R_G[r_q]}, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq w_j \leq n.$$

- B) 1. The number of paths starting in or passing through a node p_v and passing through the edge (p_v, r_w) is expressed by the path matrix $P_{(p_v, r_w)} = (V_{p_v}[-] + \sum_{R_G[r_i]=v} P_G[-, r_i]) \otimes$

$$(P_G[p_{R_G[r_w]}, -] + (V_{r_w}[-])^T), \text{ and by } R_G[r_j] = q_j \text{ for}$$

$$P_G[p_{R_G[r_w]}, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq q_j \leq n.$$

2. The number of paths starting in or passing through a node r_v and passing through the edge (r_v, p_w) is expressed by the path matrix $P_{(r_v, p_w)} = P_G[-, r_v] \otimes P_G[p_w, -]$, and by $R_G[r_j] = q_j$ for

$$P_G[p_w, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq q_j \leq n.$$

Proof: Note that $R_G[r_j] = w_j$ for $P_G[p_q, r_j] \neq 0$ expresses the paths from p_q to the nodes p_{w_j} . The number of paths to a node p_{w_j} is given

by $\sum_{R_G[r_j]=w_j} P_G[-, r_j]$. Furthermore, note in A) 2 and B) 2 that the

number of paths starting in a node r_j , $1 \leq j \leq m$, is expressed by the rows $P_G[r_j]$ in the path matrices obtained from the outer products.

The rest of the proof follows from Theorem 2 and Theorem 3. $\square$

Note that $P_{r_q} = P_{(r_q, p_w)}$ for some $p_w \in V$, since there exists only one edge $(r_q, p_w) \in E$.

Example 7: Consider the resource allocation graph G given in Example 4 and its representation P_G and R_G given in Example 5.

- A) 1. The number of paths starting in or passing through node p_2 is expressed by the path matrix $P_{p_2} = (V_{p_2}[-] + P_G[-, r_1]) \otimes$

$P_G[p_2, -]$, and by $R_G[r_j] = w_j$ for $P_G[p_2, r_j] \neq 0$, that is, by

$$r_1 r_2 r_3 r_4 r_5$$

$$P_{p_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \otimes (0 \ 0 \ 1 \ 1 \ 2) = \begin{pmatrix} 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}$$

$$w_j$$

$$R_G = \begin{bmatrix} 5 \\ 3 \\ 6 \end{bmatrix} \begin{matrix} r_3 \\ r_4 \\ r_5 \end{matrix}$$

2. The number of paths starting in or passing through node r_4 is expressed by the path matrix $P_{r_4} = P_G[-, r_4] \otimes P_G[p_{R_G[r_4]}, -]$

and by $R_G[r_j] = w_j$ for $P_G[p_{R_G[r_4]}, r_j] \neq 0$, that is, by

$$P_{r_4} = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 3 \\ 0 \\ 0 \end{bmatrix} \otimes (0 \ 0 \ 0 \ 0 \ 1) = \begin{matrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix} \end{matrix}$$

$$w_j$$

$$R_G = \begin{bmatrix} 3 \\ 6 \end{bmatrix} \begin{matrix} r_4 \\ r_5 \end{matrix}$$

B) 1. The number of paths starting in or passing through node p_2 and passing through the edge (p_2, r_4) is expressed by the path matrix $P_{(p_2, r_4)} = (V_{p_2}[-] + P_G[-, r_1]) \otimes (P_G[p_{R_G[r_4]}, -]$

$(V_{r_4}[-])^T$) and by $R_G[r_j] = q_j$ for $P_G[p_{R_G[r_4]}, r_j] \neq 0$,

that is, by

$$P_{(p_2, r_4)} = \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \right) \otimes ((0 \ 0 \ 0 \ 0 \ 1) + (0 \ 0 \ 0 \ 1 \ 0)) =$$

$$\begin{array}{c}
 r_1 r_2 r_3 r_4 r_5 \\
 \\
 = \begin{array}{c} \left(\begin{array}{ccccc} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right) \begin{array}{l} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{array} \end{array}
 \end{array}
 \begin{array}{c}
 q_j \\
 \\
 R_G = \left(\begin{array}{c} 3 \\ 6 \end{array} \right) \begin{array}{l} r_4 \\ r_5 \end{array}
 \end{array}$$

2. The number of paths starting in or passing through node r_4 and passing through the edge (r_4, p_3) is expressed by the path matrix $P_{(r_4, p_3)} = P_G[-, r_4] \otimes P_G[p_3, -]$ which is equal to

P_{r_4} given in part A) 2 and by $R_G[r_4] = 3$ and $R_G[r_5] = 6$. $\square$

3.3 Deletion and Insertion of Edges

Deletion of an edge (p_v, r_w) means that all the paths starting in or passing through node p_v and passing through (p_v, r_w) must be deleted, which can be expressed by $P_G - P_{(p_v, r_w)}$.

The deletion of an edge (r_v, p_w) can in the same manner be expressed by $P_G - P_{(r_v, p_w)}$ and by setting $R_G[r_v]$ to zero.

Example 8:

1. The deletion of the edge (p_2, r_4) from the graph G given in Example 4 is expressed by $P_{G'} = P_G - P_{(p_2, r_4)}$, and we obtain:

$$\begin{array}{c} r_1 r_2 r_3 r_4 r_5 \\ \\ \\ \\ \\ \end{array} \quad \begin{array}{c} p_i \\ \\ \\ \\ \\ \end{array}$$

$$P_{G'} = \begin{array}{c} \left(\begin{array}{ccccc} 1 & 0 & 2 & 1 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 2 & 2 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right) \begin{array}{l} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{array} \\ \\ \end{array} \quad R_{G'} = \begin{array}{c} \left(\begin{array}{c} 2 \\ 1 \\ 5 \\ 3 \\ 6 \end{array} \right) \begin{array}{l} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{array} \\ \\ \end{array}$$

2. The deletion of the edge (r_4, p_3) from the graph G given in Example 4 is expressed by $P_{G''} = P_G - P_{(r_4, p_3)}$ and by setting $R_{G''}[r_4]$ to zero. We obtain:

$$\begin{array}{c} r_1 r_2 r_3 r_4 r_5 \\ \\ \\ \\ \\ \end{array} \quad \begin{array}{c} p_i \\ \\ \\ \\ \\ \end{array}$$

$$P_{G''} = \begin{array}{c} \left(\begin{array}{ccccc} 1 & 0 & 2 & 2 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 2 & 3 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right) \begin{array}{l} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{array} \\ \\ \end{array} \quad R_{G''} = \begin{array}{c} \left(\begin{array}{c} 2 \\ 1 \\ 5 \\ 0 \\ 6 \end{array} \right) \begin{array}{l} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{array} \\ \\ \end{array}$$

□

In the same way, the insertion of an edge (p_v, r_w) can be expressed by $P_G + P_{(p_v, r_w)}$ and the insertion of an edge (r_v, p_w) can be expressed

by $P_G + P_{(r_v, p_w)}$ and by setting $R_{G''}[r_v]$ to w .

To save time when deleting or inserting an edge (p_v, r_w) (or (r_v, p_w)) we add or subtract only the non-zero elements of $P_{(p_v, r_w)}$

(or $P_{(r_v, p_w)}$). Let set S contain the non-zero elements of column

$(V_{p_v}[-] + \sum_{R_G[r_k]=v} P_G[-, r_k])$ together with their row indices and let

set Q contain the non-zero elements of row $(P_G[p_{R_G[r_w]}, -] + (V_{r_w}[-])^T)$

together with their column indices in the following algorithm:

Algorithm 1: (deletion of an edge (p_v, r_w))

```

(1)  $S := \{\}$ ;
(2) for  $i := 1$  to  $n$  do
(3)   begin
(4)      $s := 0$ ;
(5)     for all  $r_k; R_G[r_k] = v$  do
(6)       if  $P_G[p_i, r_k] \neq 0$  then
(7)          $s := s + p_G[p_i, r_k]$ ;
(8)       if  $s \neq 0$  then
(10)       $S := S \cup \{[p_i, s]\}$ 
(11)    end;
(12)  $S := S \cup \{[p_v, 1]\}$ ;          (*  $S$  = the non-zero elements      *)
                                     (* of  $(V_{p_v}[_] + \sum_{R_G[r_k]=v} P_G[_ , r_k])$  *)
(13)  $Q := \{\}$ ;
(14) for  $j := 1$  to  $m$  do
(15)   if  $P_G[p_{R_G[r_w]}, r_j] \neq 0$  then
(16)      $Q := Q \cup \{[r_j, P_G[p_{R_G[r_w]}, r_j]]\}$ ;
(17)  $Q := Q \cup \{[r_w, 1]\}$ ;          (*  $Q$  = the non-zero elements      *)
                                     (* of  $(P_G[p_{R_G[r_w]}, _] + (V_{r_w}[_])^T)$  *)
(18) for all elements  $[p_i, s]$  in  $S$  do
(19)   for all elements  $[r_j, q]$  in  $Q$  do
(20)      $P_G[p_i, r_j] := P_G[p_i, r_j] - s * q$  (*  $P_G := P_G - P_{(p_v, r_w)}$  *)

```

Theorem 6: Let $G = (V, E)$ be a resource allocation graph which is represented by its path matrix P_G and resource allocation vector R_G .

A) Let $G' = (V, E')$, $E' = E - \{(p_v, r_w)\}$, be the graph obtained after deleting the edge (p_v, r_w) from G . Algorithm 1 then correctly computes the path matrix $P_{G'}$ of the graph G' .

B) Let $G'' = (V, E'')$, $E'' = E \cup \{(p_v, r_w)\}$, be the graph obtained after inserting the edge (p_v, r_w) into G . Algorithm 1, with addition instead of subtraction in line (20), then correctly computes the path matrix $P_{G''}$ of the graph G'' .

Proof: It is easy to see that Algorithm 1 computes $P_G - P(p_v, r_w)$. The proof then follows from Theorem 3 and Theorem 5. $\square$

To express the running time of the algorithms we count the number of r- and p-nodes which can be reached from a node and the number of r- and p-nodes from which a node can be reached. Let

$\text{reach_to_r}(v) = |\{r_i; \text{there is a path of length } > 0 \text{ from } v \text{ to } r_i\}|$,
 $\text{reach_to_p}(v) = |\{p_i; \text{there is a path of length } > 0 \text{ from } v \text{ to } p_i\}|$,
 $\text{reach_from_r}(v) = |\{r_i; \text{there is a path of length } > 0 \text{ to } v \text{ from } r_i\}|$, and
 $\text{reach_from_p}(v) = |\{p_i; \text{there is a path of length } > 0 \text{ to } v \text{ from } p_i\}|$.

Theorem 7: Let the sets S and Q and the indices of the resources r_k , $k = 1, 2, \dots, m$, allocated to process p_v in Algorithm 1 be represented as linked lists. The running time of Algorithm 1 is then $O(\max(n * (\text{indeg}(p_v) + 1), m, \text{reach_from_p}(p_v) * \text{reach_to_r}(r_w)))$.

Proof: After line (12) we have $|S| = O(\text{reach_from_p}(p_v))$ and after line (17) $|Q| = O(\text{reach_to_r}(r_w))$. The running time of lines (18) - (19) is then $O(\text{reach_from_p}(p_v) * \text{reach_to_r}(r_w))$ of lines (2) - (11) $O(n * (\text{indeg}(p_v) + 1))$, of lines (14) - (16) $O(m)$ and of lines (1), (12), (13) and (17) $O(1)$. $\square$

Deletion or insertion of an edge (r_v, p_w) can be done by a slightly modified Algorithm 1 which has a running time $O(\max(n, m, \text{reach_from_p}(r_v) * \text{reach_to_r}(p_w)))$.

3.4 Deletion and Insertion of Nodes

Deleting a node p_q means that all paths ending in, starting in, or passing through p_q must be deleted. The paths ending in p_q are represented by $R_G[r_j] = q$, $1 \leq j \leq m$, and the paths starting in or passing through p_q by P_{p_q} . The deletion of p_q can then be expressed

by setting $R_G[r_j]$ to zero for all $R_G[r_j] = q$ and subtracting P_{p_q}

from P_G . This can also be done by a slightly modified Algorithm 1 which has a running_time

$$O(\max(n * (\text{indeg}(p_q) + 1), m, \text{reach_from_p}(p_q) * \text{reach_to_r}(p_q))).$$

In the same manner a node r_q can be deleted in

$$O(\max(n, m, \text{reach_from_p}(r_q) * \text{reach_to_r}(r_q))) \text{ time.}$$

Deleting a node p_q makes row p_q empty (to consist only of zeros) in the path matrix. Deleting a node r_q makes column r_q empty in the path matrix and element r_q empty in the resource allocation vector.

In the case of inserting a new node into an empty place p_q (row p_q is empty) we first compute column $(V_{p_q}[-] + \sum_{(r_v, p_q) \in E} P_G[-, r_v])$ and

row p_q . Observe that the new node obtains the name p_q corresponding to the empty place. Row p_q is given by $\sum_{(p_q, r_w) \in E} (P_G[p_{R_G[r_w]}, -]$

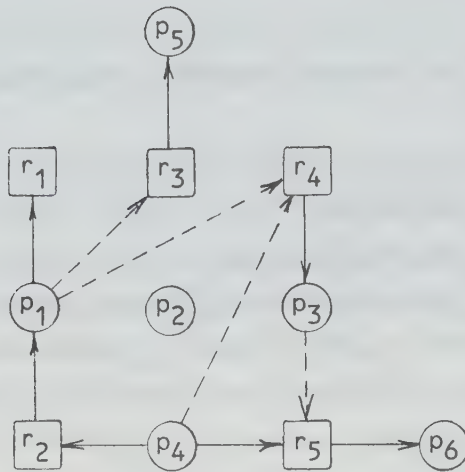
$+ (V_{r_w}[-])^T)$. Then, after computing P_{p_q} , the insertion of node p_q

is expressed by $P_G + P_{p_q}$ and setting $R_G[r_v] = q$ for the nodes r_v

from which there are edges to p_q .

Example 9:

A) Consider the resource allocation graph G given in Example 4 and its representation P_G and R_G given in Example 5 and the path matrix P_{p_2} given in Example 7. The deletion of node p_2 , expressed by $P_{G'} = P_G - P_{p_2}$ and setting $R_{G'}[r_1]$ to zero, gives the graph G' represented by:


 $r_1 r_2 r_3 r_4 r_5$
 p_i

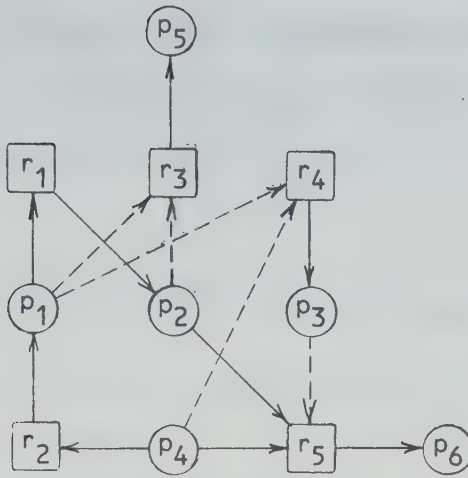
$$P_{G'} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}$$

$$R_{G'} = \begin{pmatrix} 0 \\ 1 \\ 5 \\ 3 \\ 6 \end{pmatrix} \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix}$$

B) Consider the resource allocation graph G' and its representation in part A). The insertion of a new node p_2 , represented by the edges $\langle r_1, p_2 \rangle$, $\langle p_2, r_3 \rangle$ and $\langle p_2, r_5 \rangle$, is expressed by $P_{G''} = P_{G'} + P_{p_2}$ and setting $R_{G'', [r_1]}$ to 2:

$$P_{p_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \otimes (0 \ 0 \ 1 \ 0 \ 1) = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and we obtain



$$\begin{array}{c}
 r_1 r_2 r_3 r_4 r_5 \\
 P_{G,i} = \begin{pmatrix} 1 & 0 & 2 & 1 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 2 & 2 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}
 \end{array}
 \quad
 \begin{array}{c}
 p_i \\
 R_{G,i} = \begin{pmatrix} 2 \\ 1 \\ 5 \\ 3 \\ 6 \end{pmatrix} \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix}
 \end{array}$$

□

Let set S contain the computed non-zero elements of column $(V_{p_q}[_] +$

$\sum_{(r_v, p_q) \in E} P_G[_], r_v]$) together with their row indices and let set Q

contain the computed non-zero elements of row $P_G[p_q, _]$ together with their column indices in the following algorithm:

Algorithm 2: (insertion of a node p_q)

- (1) $S := \{\}$;
- (2) for $i := 1$ to n do
- (3) begin
- (4) $p := 0$;
- (5) for all $r_v; (r_v, p_q) \in E$ do
- (6) $p := p + P_G[p_i, r_v]$;
- (7) if $p \neq 0$ then
- (8) $S := S \cup \{[p_i, p]\}$
- (9) end;

```

(10)  $S := S \cup \{[p_q, 1]\};$  (*  $S$  = the computed non-zero elements of  $*$  *)
      (*  $(V_{p_q}[-] + \sum_{(r_v, p_q) \in E} P_G[-, r_v])$  *)
(11)  $Q := \{\};$ 
(12) for  $j := 1$  to  $m$  do
(13)   begin
(14)      $t := 0;$ 
(15)     for all  $r_w; (p_q, r_w) \in E$  do
(16)       begin
(17)          $t := t + P_G[p_{R_G[r_w]}, r_j];$ 
(18)         if  $j = w$  then (* add  $(V_{r_w}[-])^T$  *)
(19)            $t := t + 1$ 
(20)       end;
(21)       if  $t \neq 0$  then
(22)          $Q := Q \cup \{[r_j, t]\}$ 
(23)       end; (*  $Q$  = the computed non-zero elements of  $P_G[p_q, -]$  *)
(24)   for all  $[p_i, p]$  in  $S$  do
(25)     for all  $[r_j, t]$  in  $Q$  do
(26)        $P_G[p_i, r_j] := P_G[p_i, r_j] + p * t;$  (*  $P_G := P_G + P_{p_q}$  *)
(27)   for all  $r_v; (r_v, p_q) \in E$  do (* update  $R_G$  *)
(28)      $R_G[r_v] := q$ 

```

Theorem 8: Let $G = (V, E)$ be a resource allocation graph which is represented by its path matrix P_G and resource allocation vector R_G .

A) Let $G' = (V', E')$, $V' = V \cup \{p_q\}$ where $p_q \notin V$, and $E' = E \cup \{(r_v, p_q), (p_q, r_w)\}$ for all $r_v, r_w \in V$, be the resource allocation graph obtained after inserting of node p_q in G . Algorithm 2 then correctly computes the path matrix $P_{G'}$ and the resource allocation vector $R_{G'}$ of the graph G' .

B) 1. Let (p_q, r_{w_j}) , $j = 1, 2, \dots, k$, be all edges starting in node p_q .

If the set S after line (9) in Algorithm 2 does not contain any element $[p_{R_G[r_{w_j}]}, p]$, $j = 1, 2, \dots, k$ and $p > 1$, then inserting

node p_q does not make the graph cyclic, that is, after inserting a process q the system will still be in a safe state.

2. Let (r_{v_j}, p_q) , $j = 1, 2, \dots, l$, be all edges ending in node p_q .

If the set Q after line (23) in Algorithm 2 does not contain any element $[r_{v_j}, t]$, $j = 1, 2, \dots, l$ and $t > 1$, then inserting node p_q does not make the graph cyclic, that is, after inserting a process q the system will still be in a safe state.

Proof:

A) It is easy to see that Algorithm 2 computes $P_G + P_{p_q}$ and updates

R_G . The proof then follows from the argument above and from Theorems 2, 3 and 5.

B) 1. Set S after line (9) contains all the nodes from which there are paths leading to p_q via the edges ending in p_q . According to Theorem 4, set S can then be used to check whether the edges starting in p_q can make the graph cyclic.

2. The proof follows in the same manner as in part 1. $\square$

Theorem 9: Let the sets S and Q be represented as linked lists and the edges ending or starting in node p_q by adjacency lists in Algorithm 2.

A) The running time of Algorithm 2 is

$$O(\max(n \cdot \text{indeg}(p_q), m \cdot \text{outdeg}(p_q), \text{reach_from_p}(p_q) \cdot \text{reach_to_r}(p_q))).$$

B) To check whether inserting the node p_q makes the resource allocation graph cyclic, that is, inserting a process q puts the system in an unsafe state, takes

$$O(\min(\max(n \cdot \text{indeg}(p_q), \text{outdeg}(p_q)), \max(m \cdot \text{outdeg}(p_q), \text{indeg}(p_q)))) \text{ time.}$$

Proof:

A) The running time of lines (2)–(9) is $O(n \cdot \text{indeg}(p_q))$ and of (12) – (23) $O(m \cdot \text{outdeg}(p_q))$. Since $|S| = O(\text{reach_from_p}(p_q))$ and $|Q| = O(\text{reach_to_r}(p_q))$ at line (25), the running time of lines (25)–(26) is $O(\text{reach_from_p}(p_q) \cdot \text{reach_to_r}(p_q))$. The running times of the other lines are less; of lines (27)–(28) $O(\text{indeg}(p_q))$ and of (1), (10) and (11) $O(1)$.

B) Let set S and Q in Algorithm 2 also have a boolean array representation with element $p_i = 1$ if $[p_i, p] \in S$ and $p > 1$ and where element $r_j = 1$ if $[r_j, t] \in Q$ and $t > 1$, respectively.

If $\text{indeg}(p_q) < \text{outdeg}(p_q)$ then compute S and the boolean representation of S in $O(n \cdot \text{indeg}(p_q))$ time, and check for cycles in

$O(\text{outdeg}(p_q))$ time. If $\text{indeg}(p_q) \geq \text{outdeg}(p_q)$ then compute Q and the boolean array representation of Q which takes $O(m \cdot \text{outdeg}(p_q))$ time, and check for cycles in $O(\text{indeg}(p_q))$ time. $\square$

Insertion of a node r_q can be done by a slightly modified Algorithm 2 in $O(\max(n \cdot \text{indeg}(r_q), m, \text{reach_from_p}(r_q) \cdot \text{reach_to_r}(r_q)))$ time. The checking whether inserting node r_q makes the graph cyclic takes $O(\min(\max(n \cdot \text{indeg}(r_q), 1), \max(m, \text{indeg}(r_q))))$ time.

3.5 Attempted Insertion of Nodes and Edges

We now give the algorithm for an attempted insertion of a node p_q , using the notations from Algorithm 2. The computations in the algorithm correspond to appropriate parts of Algorithm 2.

Algorithm 3: (attempted insertion of a node p_q)

```

(1) if  $\text{indeg}(p_q) < \text{outdeg}(p_q)$ 
    then begin
        compute  $S$ ;
        if there is a cycle
            then Exit (* unsuccessful insertion *)
            else compute  $Q$ 
        end
    else begin
        compute  $Q$ ;
        if there is a cycle
            then Exit (* unsuccessful insertion *)
            else compute  $S$ 
        end
(2) compute  $P_G := P_G + P_{p_q}$  and (* successful insertion *)
    update  $R_G$ 

```

The algorithm for attempted insertion of a node r_q has the same

structure. When attempting to insert an edge, we first check for absence of cycles in the resource allocation graph and in the case of no cycles we use Algorithm 1 (or a modification of it), with addition instead of subtraction in the line given in Theorem 6 B).

4. Conclusion

A comparison between our technique and topological sorting, which is the previously proposed technique for cycle detection, is presented in [4]. The comparison is made for acyclic digraphs when the number of edges e is $\theta(n^2)$ (where n is the number of nodes in the graph) and is relevant to resource allocation graphs. It shows that our technique is superior in the case of operations on edges; it is better by an additive term for deletion and successful insertion while for unsuccessful insertion it is much better. It gives an $O(1)$ running time while topological sorting requires $\theta(n + e)$. In the case of operations on nodes, our technique is to prefer when for each deletion and successful insertion there are more than $0.75 \cdot n$ unsuccessful insertions, which is a realistic assumption for resource allocation systems.

A simulation presented in [3] shows that rejected requests are the most frequent operations in a resource allocation system. It means that unsuccessful insertions are the most frequent of the three operations. The $O(1)$ cost of an unsuccessful insertion operation can amortize the cost of the other two types of operation. When for each deletion and successful insertion there are $O(\max(n, m))$ unsuccessful insertions, the amortized cost of one operation is linear, and when there are $O(n \cdot m)$ unsuccessful insertions, the amortized cost of one operation is constant. This latter situation may very well happen in overloaded resource allocation systems.

Our technique with its $O(1)$ cost for detection of deadlocks gives a very efficient solution of the deadlock avoidance problem. However, the technique is not well suited for deadlock detection combined with recovery from deadlock. By using a deadlock avoidance algorithm a sys-

tem is always kept in a safe state. This is possible by means of the information given by the claim edges. In the absence of such information a deadlock detection algorithm with a subsequent recovery from deadlock can be used. Note, in this case a process cannot wait for its request of a resource to be satisfied, since another process which is allocating the resource may in the future request a resource which is allocated by the first process, and then, a deadlock will occur. It means that some process which is involved in the deadlock must be aborted. In spite of the $O(1)$ cost of detecting deadlocks, the cost of deleting a node p_q makes our algorithm less suitable for this situation.

Future systems will be oriented more toward asynchronous parallel operation than toward serial systems. Multiprocessing will be common and parallel computation prevalent. There will, quite simply, be more operations going on concurrently. Resource allocation will tend to be dynamic and processes be able to acquire and release resources freely as needed. With the increasing tendency of operating systems designers to view data as a resource, the number of resources operating systems must manage will increase dramatically. In the dynamic systems of tomorrow, rejection of requests will occur very frequently. Our dynamic technique with its $O(1)$ running time for a rejection operation fits future systems very well.

Acknowledgement

The author is grateful to Dr Rolf Karlsson for his useful suggestions.

References

1. A. V. Aho & J. E. Hopcroft & J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1975.
2. A. V. Aho & J. E. Hopcroft & J. D. Ullman, "Data Structures and Algorithms", Addison-Wesley, 1983.
3. F. Belik, "Deadlock Avoidance with a Modified Banker's Algorithm", LUNFD6/(NFCS-3008), Dept. of Computer Science, Lund University, Sweden, 1985.
4. F. Belik, "Dynamic Operation on Acyclic Digraphs", LUNFD6/(NFCS-3010), Dept. of Computer Science, Lund University, Sweden, 1985.
5. P. A. Bernstein & N. Goodman, "Concurrency Control in Distributed Database Systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.
6. P. A. Bernstein & J. B. Rothnie & N. Goodman & C. A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1 : A System for Distributed Databases", IEEE Trans. on Software Engineering, Vol. SE-4, No. 3, May 1978, pp. 154-168.
7. M. C. Chen & M. Rem, "Deadlock-Freedom in Resource Contentions", Acta Informatica 21, 1985, pp. 585-598.
8. E. G. Coffman & M. J. Elphick & A. Shoshani, "System Deadlocks", Computing Surveys, Vol. 3, No. 2, June 1971, pp. 67-78.
9. H. M. Deitel, "An Introduction to Operating Systems", Addison-Wesley, 1984.
10. E. W. Dijkstra, "Cooperating Sequential Processes", Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.
11. V. D. Gligor & S. H. Shattuck, "On Deadlock Detection in Distributed Systems", IEEE Trans. on Software Engineering, Vol. SE-6, No. 5, September 1980, pp. 435-440.
12. A. N. Haberman, "Prevention of System Deadlocks", Communications of the ACM, Vol. 12, No. 7, July 1969, pp. 373-385.
13. J. W. Havender, "Avoiding Deadlock in Multitasking Systems", IBM System Journal, Vol. 7, No. 2, 1968, pp. 74-84.
14. R. C. Holt, "Some Deadlock Properties of Computer Systems", Computing Surveys, Vol. 4, No. 3, September 1972, pp. 179-196.

15. Toshihide Ibaraki & Tsunehiko Kameda, "Deadlock-Free Systems for Bounded Number of Processes", IEEE Trans. on Computers, Vol. C-31, No. 3, March 1982, pp. 188-193.
16. Tiko Kameda, "Testing Deadlock-Freedom of Computer Systems", Journal of the ACM, Vol.27, No. 2, April 1980, pp. 270-280.
17. D. E. Knuth, "The Art of Computer Programming", Volume 1, Addison-Wesley, 1975.
18. W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.
19. J. Y.-T. Leung & E. K. Lai, "On Minimum Cost Recovery from System Deadlock", IEEE Trans. on Computers, Vol. C-28, No. 9, September 1979, pp. 671-677.
20. D. B. Lomet, "Subsystems of Processes with Deadlock Avoidance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 297-304.
21. H. Madduri & R. Finkel, "Extension of the Banker's Algorithm for Resource Allocation in a Distributed Operating System", Inf. Process. Lett. 19, 1 (July 1984), pp. 1-8.
22. K. Mehlhorn, "Graph Algorithms and NP-Completeness", Springer Verlag, 1984.
23. D. A. Menasce & R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 195-202.
24. J. Peterson & A. Silberschatz, "Operating System Concepts", Addison-Wesley, 1985.
25. D. J. Rypka & A. P. Lucido, "Deadlock Detection and Avoidance for Shared Logical Resources", IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, January 1985, pp. 67-80.
26. M. K. Sinha & N. Natarajan, "A Priority Based Distributed Deadlock Detection Algorithm", IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, January 1985, pp. 67-80.
27. M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed I N G R E S", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 188-194.
28. R. E. Tarjan, "Data Structures and Network Algorithms", SIAM Monograph, 1983.

AN EFFICIENT TECHNIQUE TO AVOID DEADLOCK IN DISTRIBUTED SYSTEMS

Ferenc Belik

Department of Computer Science,
Lund University, Sweden

Abstract

We present a dynamic technique which is considerably better for detecting deadlocks, when allocating a resource in distributed systems with single instances of each resource type, than topological sorting, the previously proposed technique. Furthermore, our technique gives a very efficient solution to the deadlock avoidance problem in a fully distributed environment. We consider the deadlock avoidance problem as the problem of changing a directed acyclic graph while keeping it acyclic. The resource allocation algorithms then involve three operations on edges or on nodes: deletion, unsuccessful insertion and successful insertion, where an insertion first checks if a cycle would result. The resource allocation graph is represented by a path matrix which is easy to partition so that each site contains the necessary information to efficiently detect cycles. In resource allocation systems rejected requests, that is, unsuccessful insertion of edges, are the most frequent operations. Our dynamic technique detects a cycle (i.e., the insertion is unsuccessful) in $O(1)$ time when operating on edges, so our technique uses only $O(1)$ time to detect deadlock when the system allocates a constant number of resources to a process. This $O(1)$ cost of the unsuccessful insertion operation can amortize the cost of the other two types of operation and linear (or even constant) amortized time for one operation per site is realistic in resource allocation systems. The number of sent messages can also be amortized depending on the speed of the message traffic. With sufficiently high speed an amortized constant number of sent messages is realistic per operation.

A comparison with topological sorting, when the number of edges is $\theta(n^2)$, shows that our technique is superior. It is better by an additive term when allocating a resource and it is much better when rejecting a resource request, requiring only $O(1)$ time while topological sorting needs $\theta(n^2)$.

1. Introduction

As more distributed processing systems are implemented the requirement for distributed deadlock handling becomes more apparent. A situation called deadlock was pointed out by Dijkstra [12] in a computer system in which two or more processes coexist sharing resources. A simple example is where a process P_1 holds a device A and requires device B to continue, but device B is already held by another process P_2 which require A to continue. There is a circular wait here and it will not be resolved unless a "drastic" action, i.e., preemption is taken. Informally, a deadlock is a situation in which there is a process whose progress is blocked forever owing to lack of resources, regardless of the future activity (e.g., release of resource unit) of unblocked processes. Different aspects of the deadlock problem have been investigated in [3,8,10,14-18,22,31], and new interest in it has been revived in the database and distributed environment [6,7,13,20,23,24,26,30,32,33]. Two necessary and sufficient conditions for the absence of deadlock were established under the expedient allocation policy in [9]. Coffman et al. [10] classify approaches dealing with deadlocks into three categories: detection and recovery, avoidance and prevention. This paper considers a deadlock avoidance algorithm in systems with single instances of each resource type in distributed environment. The resource allocation states in such systems can be described by resource allocation graphs with the deadlock situations expressed by cycles in the graphs. Topological sorting is an efficient algorithm used for the detection of cycles in directed graphs [1,2,19,25,35] and it is the previously proposed technique for detection of deadlocks. The algorithm uses a graph reduction technique with an adjacency list representation of the graph and may scan the whole graph to find a cycle. The running time of the algorithm is $O(n + e)$, where n is the number of nodes and e is the

number of edges in the graph. However, it is wasteful to scan the whole graph as topological sorting does. In resource allocation systems only a few resources (in most cases only one resource) are allocated or released to or from a process at a time. It means that the resource allocation graph is changed by insertion or deletion of a few edges. Further, the graph may also be changed by insertion or deletion of nodes (that is, processes or resources). The resource allocation problem can then be considered as the problem of changing an acyclic digraph while keeping it acyclic by involving three operations on edges or nodes. These operations are: deletion, unsuccessful insertion and successful insertion where the insertions include a previous "detection" of cycles. In this paper we propose a dynamic technique which efficiently solves the problem. In resource allocation systems rejected requests, that is, unsuccessful insertions of edges, are the most frequent operations [3]. Our dynamic technique, using a path matrix representation of the resource allocation graph, gives $O(1)$ time for detection of cycles (i.e., for unsuccessful insertions) when operating on edges. That is, our technique uses only $O(1)$ time for detection of deadlocks when the system allocates a constant number of resources to a process. Thus, this $O(1)$ cost can amortize the cost of the other two types of operation and linear (or even constant) amortized time for one operation is realistic in resource allocation systems. By amortized time we mean the time of an operation averaged over a worst-case sequence of operations. A comparison between our technique and topological sorting is given in [4].

The application of the dynamic technique to tightly coupled multiprocessor systems was given in [5]. In this paper we propose an application of the technique to distributed systems. The problem of deadlock avoidance in a distributed environment is rarely dealt with in the literature. The reason for this may be that the necessary a priori information to avoid deadlocks makes the resource allocation graph more complex. We quote: "this approach is not practical in distributed data bases since the necessary advance information to avoid deadlocks is either absent or distributed enough to render inefficient any attempt to avoid deadlocks" [26]. However, our technique gives full control over the distributed information and produce a very efficient solution to the deadlock avoidance problem in a fully distributed environment. In Section 2 we give the basic definitions and the representation of a directed acyclic graph. Section 3 contains the representation of the resource allocation graph, the description of the dynamic technique

used in the resource allocation algorithms and the operations on edges and nodes. The applications of the technique to centralized and fully distributed approaches are given in Section 4.1 and 4.2 respectively.

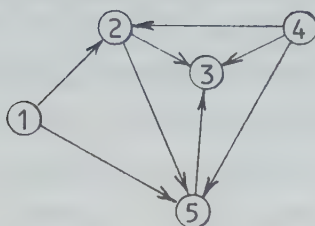
2. Definitions and Representation of an Acyclic Digraph

We define a directed graph (or digraph) $G = (V, E)$ consisting of a set $V = \{1, 2, \dots, |V|\}$ of nodes and a set $E \subseteq V \times V$ of edges. A pair $(v, w) \in E$ is called an edge from v to w . A path from v to w , $v, w \in V$, is a sequence $v_0, v_1, v_2, \dots, v_k$ of nodes such that $v_0 = v$, $v_k = w$ and $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$; k is the length of the path. Note that there is always the path of length zero from v to v . A cycle is a path from v to v . A graph is acyclic if it contains no non-trivial cycles. The indegree of a node v is the number of edges ending in v , $\text{indeg}_G(v) = |\{w ; (w, v) \in E\}|$. Similarly, the outdegree of v is the number of edges starting in v , $\text{outdeg}_G(v) = |\{w ; (v, w) \in E\}|$.

An appropriate representation of a graph for edgewise detection of cycles and for insertion and deletion of edges is a path matrix, which is a $|V| \times |V|$ matrix $P_G = (p_{ij})_{1 \leq i, j \leq n}$ with

$$p_{ij} = \begin{cases} m & \text{if there are } m \geq 1 \text{ different paths of length } > 0 \\ & \text{from node } i \text{ to node } j \\ 0 & \text{if there is no path of length } > 0 \text{ from node } i \text{ to} \\ & \text{node } j \end{cases}$$

The storage requirement of this representation is clearly $\theta(n^2)$.

Example_1: A graph G and its path matrix

$$P_G = \begin{pmatrix} 0 & 1 & 3 & 0 & 2 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

□

To check if the insertion of an edge (v,w) into a graph G makes the graph cyclic it is sufficient to have a set $P = \{ [v_i, v_j] \}$; there is a path of length > 0 from v_i to v_j , $v_i, v_j \in V$. If P is represented as a boolean matrix it takes $O(1)$ time to check if $P[w,v] \neq 0$. However, this representation is not enough if we want to be able to delete edges. Instead consider the path matrix representation given above. It is clear that the path matrix is suitable for the detection of cycles in $O(1)$ time. To make the description of the other operations easier we will take a closer look at the path matrix representation.

Theorem_1: Let $G = (V,E)$ be an acyclic digraph and let P_G be its path matrix representation as defined above.

- A) Consider node q with edges (r,q) , $r = i,j,\dots,l$ ending in q . Then, column q of the path matrix P_G , denoted by $P_G[_,q]$, can be partitioned into $P_G[_,i], P_G[_,j], \dots, P_G[_,l], V_i[_,], V_j[_,], \dots, V_l[_,]$, where $P_G[_,r]$ denotes column r in P_G and $V_r[_,]$, $r = i,j,\dots,l$ are column vectors of length n , consisting of zero elements anywhere except at index r which is set to one. The vector $V_r[_,]$ corresponds to the immediate path from r to q via the edge (r,q) .
- B) Consider node q with edges (q,r) , $r = i,j,\dots,m$ starting in q . Then, row $P_G[q, _]$ can be partitioned into $P_G[i, _], P_G[j, _], \dots, P_G[m, _], (V_i[_,])^T, (V_j[_,])^T, \dots, (V_m[_,])^T$ where T denotes transpose. The vector $(V_r[_,])^T$ corresponds to the immediate path from q to r via the edge (q,r) .
- C) The path matrix P_G is a unique representation of the graph G .

Proof:

- A) Take an element p_{kq} of column $P_G[_,q]$. A value s at p_{kq} means that

there are s paths of length > 0 from node k to node q .

If $\text{indeg}_G(q) = 0$ then $s = 0$. Otherwise the theorem says that s can be partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}, v_i[k], v_j[k], \dots, v_l[k]$, where $v_r[k] = 0$ if $r = k$. We have two cases:

1) If $s = 0$ then there is no path from k to q , the edge (k, q) cannot exist, which means that $v_k[-]$ does not exist.

Furthermore, for all $r = i, j, \dots, l$, p_{kr} must be zero, that is, there is no path from k to r . Similary the converse is true, that is, if there is no path from k to q then the partitioning of p_{kq} must consist of zeros.

2) There are $s > 0$ paths from k to q . If the edge (k, q) exists then $v_k[k] = 1$ and $s - 1$ is partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}$ since p_{kr} , $r = i, j, \dots, l$, is the number of paths from k to r and there is one path from r to q . On the other hand, if there is no edge (k, q) then $v_k[-]$ does not exist and s is partitioned into $p_{ki}, p_{kj}, \dots, p_{kl}$. The converse is also true, that is, the sum of $p_{ki}, p_{kj}, \dots, p_{kl}$ is s or $s - 1$ depending on whether there exists an edge from k to q .

B) The proof follows in the same way as in part A.

C) Since the partitioning is unambiguous the path matrix P_G correctly represents the graph G . $\square$

Example 2: The column $P_G[-, 3]$ in the path matrix given in Example 1 may be partitioned into $P_G[-, 2], P_G[-, 4], P_G[-, 5], v_2[-], v_4[-], v_5[-]$, that is,

$$P_G[-, 3] = \begin{pmatrix} 3 \\ 2 \\ 0 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 0 \\ 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \quad \square$$

Henceforth, we will use the notations introduced in Theorem 1, and in context with a path and a node or an edge, we will use the words: starting, ending and passing through. We define the paths starting in a node q as $\{v_0, v_1, \dots, v_k; v_0 = q \text{ and } (v_i, v_{i+1}) \in E\}$, the paths ending in q as $\{v_0, v_1, \dots, v_k; v_k = q \text{ and } (v_i, v_{i+1}) \in E\}$, the paths

passing through q as $\{v_0, v_1, \dots, v_k; v_i = q \text{ for some } i \neq 0, k \text{ and } (v_i, v_{i+1}) \in E\}$ and the paths passing through an edge (v, w) as $\{v_0, v_1, \dots, v_k; v_j = v, v_{j+1} = w \text{ where } 0 \leq j < k \text{ and } (v_i, v_{i+1}) \in E\}$. We will use the expression number_of_paths to express the size of these sets. To describe the number of paths passing through a node or an edge we need the definition of the outer product of a column and a row vector. Let V_c be a column vector and V_r a row vector, both of length n . The outer product, denoted by $V_c \otimes V_r$, is a $|V_c| \times |V_r|$ matrix $P = (p_{ij})_{1 \leq i, j \leq n}$ with $p_{ij} = V_c[i] * V_r[j]$.

Theorem 2: Let $G = (V, E)$ be an acyclic digraph and P_G its path matrix representation.

- A) The number of paths starting in or passing through a node q is expressed by the path matrix $P_q = (P_G[_ , q] + V_q[_]) \otimes P_G[q, _]$.
- B) The number of paths starting in or passing through a node v and passing through the edge (v, w) is expressed by the path matrix $P_{(v, w)} = (P_G[_ , v] + V_v[_]) \otimes (P_G[w, _] + (V_w[_])^T)$.

Proof:

- A) The number of paths ending in q is expressed by $P_G[_ , q]$ and the starting of paths in q by $V_q[_]$. The proceeding of the paths ending in q corresponds to the paths passing through q . The nodes reachable from q are given by the non-zero elements in row $P_G[q, _]$. The number of paths from q to a node w is given by $P_G[q, w]$. According to Theorem 1, $P_G[q, w] = k$ paths from q to node w means that the paths, represented by columns $P_G[_ , q]$ and $V_q[_]$, enter node w k times. Then, Theorem 1 and the definition of the outer product give the proof.
- B) Note that the paths from v proceed to w and to the nodes corresponding to the non-zero elements of row $P_G[w, _]$. This is expressed by the row vector $(P_G[w, _] + (V_w[_])^T)$ in the outer product. The proof then follows in the same manner as in part A). $\square$

Example 3: Consider graph G and its path matrix P_G given in Example 1.

- A) The number of paths starting in or passing through node 2 is expressed by the path matrix $P_2 = (P_G[_ , 2] + V_2[_]) \otimes P_G[2, _]$, that is, by

$$P_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \otimes (0 \ 0 \ 2 \ 0 \ 1) = \begin{bmatrix} 0 \ 0 \ 2 \ 0 \ 1 \\ 0 \ 0 \ 2 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 2 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \ 0 \end{bmatrix} \quad \square$$

Note that it is not immediate from the path matrix if there exist an edge (v,w) (i.e., a path v,w). To see that, we have to use an algorithm with a running time of $O(n)$ [4].

3. Deadlock Avoidance

In order to avoid deadlock in a system, we need to have some additional information on how each process will request resources. There are various models that differ in the amount of such information provided. The simplest and most useful model requires that each process declare all the resources that it may need. Given such a priori information it is possible to construct an algorithm that will ensure the system never to enter a deadlock state. This algorithm defines the deadlock avoidance approach.

3.1 The Resource Allocation Graph and its Representation

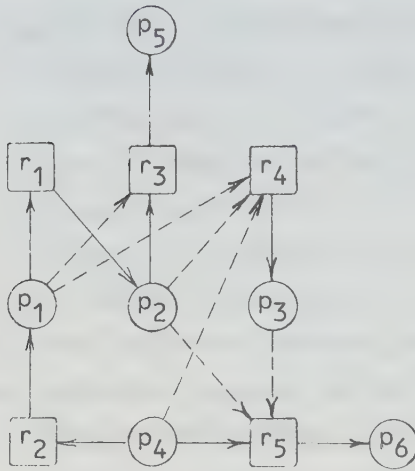
Deadlocks can be described more precisely in terms of a directed bipartite graph called a system resource allocation graph. This graph consists of a pair $G = (V,E)$ where V is a set of vertices and E is a set of edges. The set of vertices is partitioned into two types $P = \{p_1, p_2, \dots, p_n\}$, the set consisting of all processes in the system,

and $R = \{r_1, r_2, \dots, r_m\}$, the set consisting of all resources in the system.

Each element in the set E of edges is a ordered pair (p_i, r_j) or (r_j, p_i) , where p_i is a process ($p_i \in P$) and r_j is a resource ($r_j \in R$). If $(p_i, r_j) \in E$, then there is a directed edge from process p_i to resource r_j implying that process p_i requested resource r_j and is currently waiting for that resource. If $(r_j, p_i) \in E$, then there is a directed edge from resource r_j to process p_i implying that resource r_j has been allocated to process p_i . An edge (p_i, r_j) is called a request edge, while an edge (r_j, p_i) is called an assignment edge. In addition to the request and assignment edges, we have a third type of edge called a claim edge. A claim edge $\langle p_i, r_j \rangle$ indicates that process p_i may request resource r_j sometime in the future. This edge resembles a request edge in direction, but is represented by a dashed line in figures. When process p_i requests resource r_j , the claim edge $\langle p_i, r_j \rangle$ is converted to a request edge. When resource r_j is allocated to process p_i the request edge (p_i, r_j) is converted to an assignment edge (r_j, p_i) , and similarly, when a resource r_j is released by p_i , the assignment edge (r_j, p_i) is reconverted to a claim edge $\langle p_i, r_j \rangle$ if the process may request the resource again in the future, otherwise the edge can be deleted. We note that the resources must be claimed a priori in the system. That is, before process p_i starts executing, all of its claim edges must already appear in the resource allocation graph.

Pictorially, we will represent each process p_i as a circle and each resource r_j as a square.

Example 4: A resource allocation graph G and its path matrix:



$p_1 p_2 p_3 p_4 p_5 p_6$	$r_1 r_2 r_3 r_4 r_5$
0 1 2 0 2 3	1 0 2 2 3 p_1
0 0 1 0 1 2	0 0 1 1 2 p_2
0 0 0 0 0 1	0 0 0 0 1 p_3
1 1 3 0 2 5	1 1 2 3 5 p_4
0 0 0 0 0 0	0 0 0 0 0 p_5
0 0 0 0 0 0	0 0 0 0 0 p_6
0 1 1 0 1 2	0 0 1 1 2 r_1
1 1 2 0 2 3	1 0 2 2 3 r_2
0 0 0 0 1 0	0 0 0 0 0 r_3
0 0 1 0 0 1	0 0 0 0 1 r_4
0 0 0 0 0 1	0 0 0 0 0 r_5

☐

We can save both time and storage by improving the representation given in Section 2 and used in Example 4 above. The row p_i in the path matrix gives the number of paths from the node corresponding to process p_i to all nodes in the resource allocation graph. Note that there exists only one edge (r_j, p_i) for each resource r_j , $j = 1, 2, \dots, m$ and no edges between processes or between resources.

Then, there are almost the same number of paths from r_j to the other nodes as from p_i , that is, the rows r_j and p_i are almost equal. The only difference is that element p_i in row p_i is zero while in row r_j it is one, corresponding to the path r_j, p_i . Thus, it is sufficient to store the rows $p_1, p_2, \dots, p_n$, and in a vector R_G store the edges (r_j, p_i) , $j = 1, 2, \dots, m$. Furthermore, the path matrix can be further reduced by storing only the columns $r_1, r_2, \dots, r_m$. This representation would correctly represent the resource allocation graph since the number of paths leading to p_i via the edges (r_j, p_i) are given by the sum of the columns $P_G[_, r_j]$, where $R_G[r_j] = i$. In this way we obtain the condensed representation of the resource allocation graph consisting of a path matrix, which is an $n \times m$ matrix

$$P_G = (p_{p_i r_j})_{1 \leq i \leq n, 1 \leq j \leq m} \text{ with}$$

$$p_{p_i r_j} = \begin{cases} k & \text{if there are } k \geq 1 \text{ different paths of length } > 0 \\ & \text{from node } p_i \text{ to node } r_j \\ 0 & \text{if there is no path of length } > 0 \text{ from node } p_i \text{ to} \\ & \text{node } r_j \end{cases}$$

and of a resource allocation vector, which is a m - vector

$R_G = (r_j)_{1 \leq j \leq m}$ with

$$r_j = \begin{cases} i & \text{if resource } j \text{ is allocated to process } p_i \\ 0 & \text{if resource } j \text{ is not allocated to any process} \end{cases}$$

The storage requirement of this representation is $\theta(n*m)$.

Example 5: The condensed representation of the graph G given in Example 4 is:

$$P_G = \begin{matrix} & r_1 r_2 r_3 r_4 r_5 \\ \begin{pmatrix} 1 & 0 & 2 & 2 & 3 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 2 & 3 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix} \end{matrix} \quad R_G = \begin{matrix} & p_i \\ \begin{pmatrix} 2 \\ 1 \\ 5 \\ 3 \\ 6 \end{pmatrix} & \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix} \end{matrix}$$

□

The results of Theorem 1 and Theorem 2 are also valid for the condensed representation of the resource allocation graph.

Theorem 3: Let $G = (V, E)$ be a resource allocation graph represented by its path matrix P_G and resource allocation vector R_G as defined above.

A) 1. Consider node r_j with edges (p_i, r_j) , $i = a, b, \dots, f$, ending in

r_j . Then, column $P_G[-, r_j]$ of the path matrix P_G can be partitioned into

$$P_G[-, r_k]_{R_G[r_k]=a}, P_G[-, r_k]_{R_G[r_k]=b}, \dots, P_G[-, r_k]_{R_G[r_k]=f},$$

$$V_{p_a}[-], V_{p_b}[-], \dots, V_{p_f}[-]$$

where columns $P_G[-, r_k]$ are different and $V_{p_i}[-]$, $i = a, b, \dots, f$,

are column vectors of length n consisting of zero elements everywhere except at index p_i which is set to one.

2. Consider node p_i with edges (p_i, r_j) , $j = a, b, \dots, g$, starting in p_i . Then, row $P_G[p_i, -]$ can be partitioned into

$$P_G[p_{R_G[r_a]}, -], P_G[p_{R_G[r_b]}, -], \dots, P_G[p_{R_G[r_g]}, -],$$

$$(V_{r_a}[-])^T, (V_{r_b}[-])^T, \dots, (V_{r_g}[-])^T$$

where $V_{r_j}[-]$ are column vectors of length m consisting of zero

elements everywhere except at index r_j which is set to one.

- B) The path matrix P_G and the resource allocation vector R_G gives an unique representation of the graph G .

Proof:

- A) Note in case 1 that the columns $P_G[-, r_k]$, $R_G[r_k] = a, b, \dots, f$, are different since a resource can only be allocated to one process.

Furthermore, in case 2, the non-condensed representation of the resource allocation graph, $P_G[p_i, -]$ can be partitioned into

$$P_G[r_a, -], P_G[r_b, -], \dots, P_G[r_g, -],$$

$$(V_{r_a}[-])^T, (V_{r_b}[-])^T, \dots, (V_{r_g}[-])^T$$

according to Theorem 1; and a row $P_G[r_j, -]$ in the non-condensed representation is equal to $P_G[p_{R_G[r_j]}, -] + (V_{p_{R_G[r_j]}}[-])^T$ in

the condensed representation according to the discussion above, where $V_{p_{R_G[r_j]}}[-]$ and $V_{r_j}[-]$ are column vectors of length $m+n$

consisting of zero elements everywhere except at index $p_{R_G[r_j]}$ and

r_j respectively, which are set to one. However, the partitioning does not contain the vectors $V_{p_{R_G[r_j]}}[-]$ since the condensed representation does not contain the columns $p_{R_G[r_j]}$, $j = a, b, \dots, g$.

The rest of the proof follows from the discussion above and from Theorem 1.

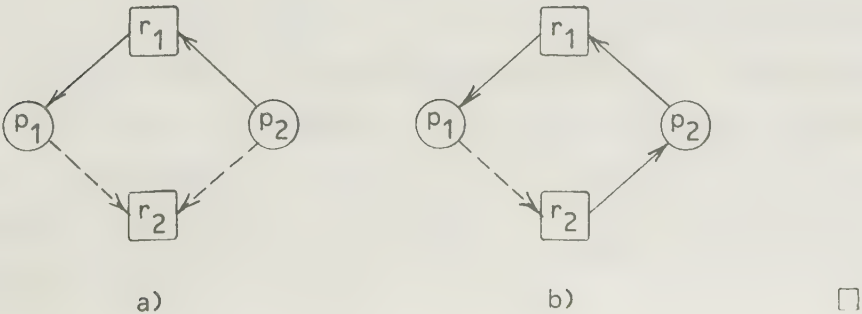
B) The proof follows from part A) and Theorem 1. □

Henceforth, we will also use the notations from Theorem 3.

3.2 The Dynamic Technique Used in the Resource Allocation Algorithm

Suppose that process p_i requests resource r_j . The request can be granted only if reversing the request edge $\langle p_i, r_j \rangle$ to an assignment edge (r_j, p_i) does not result in a cycle in the resource allocation graph. The safety check is done by a cycle detection algorithm. If no cycle exists, the allocation of the resource will leave the system in a safe state. Conversely if a cycle is found, the process p_i have to wait for its request to be satisfied since the allocation would put the system in an unsafe state. To illustrate the algorithm, consider the resource allocation graph of Example 6 a) and suppose that p_2 requests r_2 . Although r_2 is currently free, we cannot allocate it to p_2 since this action will create a cycle in the graph (Example 6 b)). A cycle indicates that the system is in an unsafe state since if p_1 then requests r_2 , a deadlock will occur.

Example 6: A safe and an unsafe state in a simple system:



More formally, we define a resource allocation system to be in an unsafe state if the resource allocation graph contains a cycle. If there is no cycle in the resource allocation graph then we say the system is in a safe state.

The resource allocation algorithm can be considered as an algorithm which changes the resource allocation graph while keeping it acyclic. There are three operations in the algorithm on edges or on nodes: deletion, unsuccessful insertion, and successful insertion, where the insertions include a previous detection of cycles. Operations on edges correspond to resource allocation while operations on nodes correspond to insertion (or deletion) of processes or resources to (from) the system. An unsuccessful insertion means that the requesting process has to wait for its request(s) to be satisfied. In the following, we do not differentiate between a claim edge and a request edge, that is, we assume that a process needs all its claimed resources to finish its task.

Our representation of a resource allocation graph makes it possible to detect in constant time whether the insertion of an edge (r_v, p_w) leads to an unsafe state of the system.

Theorem 4: Let a resource allocation system be described by a resource allocation graph $G = (V, E)$, represented by a path matrix P_G and a resource allocation vector R_G . Then the allocation of resource r_v to process p_w , that is, the insertion of the edge (r_v, p_w) into G , may put the system in an unsafe state if and only if $P_G[p_w, r_v] > 1$. The cost of this operation is $O(1)$.

Proof: We assume that $R_G[r_v] = 0$, that is, resource r_v is not allocated to any process, it can otherwise be checked in $O(1)$ time.

We first prove the correctness. It is clear that the insertion of an edge (j, i) into an acyclic digraph makes the graph cyclic if and only if there exists a path from i to j . In our resource allocation system a resource is always claimed or requested in advance, that is,

there always exist a path p_w, r_v in G corresponding to the edge (p_w, r_v) when process p_w requests resource r_v . Therefore, the insertion of the edge (r_v, p_w) puts the system in an unsafe state if and only if $P_G[p_w, r_v] > 1$. It is clear that the checking whether $P_G[p_w, r_v] > 1$ takes $O(1)$ time. $\square$

The paths passing through a node or an edge in the resource allocation graph can also be expressed as an outer product.

Theorem 5: Let $G = (V, E)$ be a resource allocation graph and let it be represented by the path matrix P_G and the resource allocation vector R_G .

- A) 1. The number of paths starting in or passing through a node p_q is expressed by the path matrix

$$P_{p_q} = (V_{p_q}[-] + \sum_{R_G[r_j]=q} P_G[-, r_j]) \otimes P_G[p_q, -], \text{ and by}$$

$$R_G[r_j] = w_j \text{ for } P_G[p_q, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq w_j \leq n.$$

2. The number of paths starting in or passing through a node r_q is expressed by the path matrix $P_{r_q} = P_G[-, r_q] \otimes P_G[p_{R_G[r_q]}, -],$

$$\text{and by } R_G[r_j] = w_j \text{ for } P_G[p_{R_G[r_q]}, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq w_j \leq n.$$

- B) 1. The number of paths starting in or passing through a node p_v and passing through the edge (p_v, r_w) is expressed by the path matrix $P_{(p_v, r_w)} = (V_{p_v}[-] + \sum_{R_G[r_j]=v} P_G[-, r_j]) \otimes$

$$(P_G[p_{R_G[r_w]}, -] + (V_{r_w}[-])^T), \text{ and by } R_G[r_j] = q_j \text{ for}$$

$$P_G[p_{R_G[r_w]}, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq q_j \leq n.$$

2. The number of paths starting in or passing through a node r_v and passing through the edge (r_v, p_w) is expressed by the path matrix $P_{(r_v, p_w)} = P_G[-, r_v] \otimes P_G[p_w, -],$ and by $R_G[r_j] = q_j$ for

$$P_G[p_w, r_j] \neq 0, j = 1, 2, \dots, m \text{ and } 1 \leq q_j \leq n.$$

Proof: Note that $R_G[r_j] = w_j$ for $P_G[p_q, r_j] \neq 0$ expresses the paths from p_q to the nodes p_{w_j} . The number of paths to a node p_{w_j} is given

by $\sum_{R_G[r_i]=w_j} P_G[-, r_i]$. Furthermore, note in A) 2 and B) 2 that the

number of paths starting in a node r_j , $1 \leq j \leq m$, is expressed by the rows $p_{R_G[r_j]}$ in the path matrices obtained from the outer products.

The rest of the proof follows from Theorem 2 and Theorem 3. $\square$

Note that $P_{r_q} = P_{(r_q, p_w)}$ for some $p_w \in V$, since there exists only one edge $(r_q, p_w) \in E$.

Example 7: Consider the resource allocation graph G given in Example 4 and its representation P_G and R_G given in Example 5.

A) 1. The number of paths starting in or passing through node p_2 is expressed by the path matrix $P_{p_2} = (V_{p_2}[-] + P_G[-, r_1]) \circledast$

$P_G[p_2, -]$, and by $R_G[r_j] = w_j$ for $P_G[p_2, r_j] \neq 0$, that is, by

$$P_{p_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \circledast (0 \ 0 \ 1 \ 1 \ 2) = \begin{matrix} \begin{pmatrix} 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix} \end{matrix}$$

w_j

$$R_G = \begin{pmatrix} 5 \\ 3 \\ 6 \end{pmatrix} \begin{matrix} r_3 \\ r_4 \\ r_5 \end{matrix}$$

2. The number of paths starting in or passing through node r_4 is expressed by the path matrix $P_{r_4} = P_G[-, r_4] \circledast P_G[p_{R_G[r_4]}, -]$

and by $R_G[r_j] = w_j$ for $P_G[p_{R_G[r_4]}, r_j] \neq 0$, that is, by

$$\begin{array}{c} r_1 r_2 r_3 r_4 r_5 \\ P_{r_4} = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 3 \\ 0 \\ 0 \end{pmatrix} \otimes (0 \ 0 \ 0 \ 0 \ 1) = \begin{pmatrix} 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}
 \end{array}$$

$$w_j$$

$$R_G = \begin{pmatrix} 3 \\ 6 \end{pmatrix} \begin{matrix} r_4 \\ r_5 \end{matrix}$$

B) 1. The number of paths starting in or passing through node p_2 and passing through the edge (p_2, r_4) is expressed by the path matrix $P_{(p_2, r_4)} = (V_{p_2}[-] + P_G[-, r_1]) \otimes (P_G[p_{R_G[r_4]}, -]$

$(V_{r_4}[-])^T$) and by $R_G[r_j] = q_j$ for $P_G[p_{R_G[r_4]}, r_j] \neq 0$,

that is, by

$$P_{(p_2, r_4)} = \left(\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \right) \otimes ((0 \ 0 \ 0 \ 0 \ 1) + (0 \ 0 \ 0 \ 1 \ 0)) =$$

$$r_1 r_2 r_3 r_4 r_5$$

$$= \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}$$

$$q_j$$

$$R_G = \begin{pmatrix} 3 \\ 6 \end{pmatrix} \begin{matrix} r_4 \\ r_5 \end{matrix}$$

2. The number of paths starting in or passing through node r_4 and passing through the edge (r_4, p_3) is expressed by the path matrix $P_{(r_4, p_3)} = P_G[-, r_4] \otimes P_G[p_3, -]$ which is equal to

$$P_{r_4} \text{ given in part A) 2 and by } R_G[r_4] = 3 \text{ and } R_G[r_5] = 6. \quad \square$$

3.3 Deletion and Insertion of Edges

Deletion of an edge (p_v, r_w) means that all the paths starting in or passing through node p_v and passing through (p_v, r_w) must be deleted, which can be expressed by $P_G - P_{(p_v, r_w)}$.

The deletion of an edge (r_v, p_w) can in the same manner be expressed by $P_G - P_{(r_v, p_w)}$ and by setting $R_G[r_v]$ to zero.

Example 8:

1. The deletion of the edge (p_2, r_4) from the graph G given in Example 4 is expressed by $P_{G'} = P_G - P_{(p_2, r_4)}$, and we obtain:

$$\begin{array}{c}
 \begin{array}{ccccc} r_1 & r_2 & r_3 & r_4 & r_5 \end{array} \\
 P_{G'} = \left[\begin{array}{ccccc|c} 1 & 0 & 2 & 1 & 2 & p_1 \\ 0 & 0 & 1 & 0 & 1 & p_2 \\ 0 & 0 & 0 & 0 & 1 & p_3 \\ 1 & 1 & 2 & 2 & 4 & p_4 \\ 0 & 0 & 0 & 0 & 0 & p_5 \\ 0 & 0 & 0 & 0 & 0 & p_6 \end{array} \right]
 \end{array}
 \quad
 \begin{array}{c}
 \begin{array}{c} p_i \end{array} \\
 R_{G'} = \left[\begin{array}{c|c} 2 & r_1 \\ 1 & r_2 \\ 5 & r_3 \\ 3 & r_4 \\ 6 & r_5 \end{array} \right]
 \end{array}$$

2. The deletion of the edge (r_4, p_3) from the graph G given in Example 4 is expressed by $P_{G''} = P_G - P_{(r_4, p_3)}$ and by setting $R_G[r_4]$ to zero. We obtain:

$$\begin{array}{c}
 r_1 r_2 r_3 r_4 r_5 \\
 P_{G,i} = \begin{pmatrix} 1 & 0 & 2 & 2 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 2 & 3 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}
 \end{array}
 \quad
 \begin{array}{c}
 p_i \\
 R_{G,i} = \begin{pmatrix} 2 \\ 1 \\ 5 \\ 0 \\ 6 \end{pmatrix} \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix}
 \end{array}
 \quad \square$$

In the same way, the insertion of an edge (p_v, r_w) can be expressed by $P_G + P_{(p_v, r_w)}$ and the insertion of an edge (r_v, p_w) can be expressed by $P_G + P_{(r_v, p_w)}$ and by setting $R_G[r_v]$ to w .

3.4 Deletion and Insertion of Nodes

Deleting a node p_q means that all paths ending in, starting in, or passing through p_q must be deleted. The paths ending in p_q are represented by $R_G[r_j] = q$, $1 \leq j \leq m$, and the paths starting in or passing through p_q by P_{p_q} . The deletion of p_q can then be expressed

by setting $R_G[r_j]$ to zero for all $R_G[r_j] = q$ and subtracting P_{p_q}

from P_G . Deleting a node p_q makes row p_q empty (to consist only of zeros) in the path matrix. Deleting a node r_q makes column r_q empty in the path matrix and element r_q empty in the resource allocation vector.

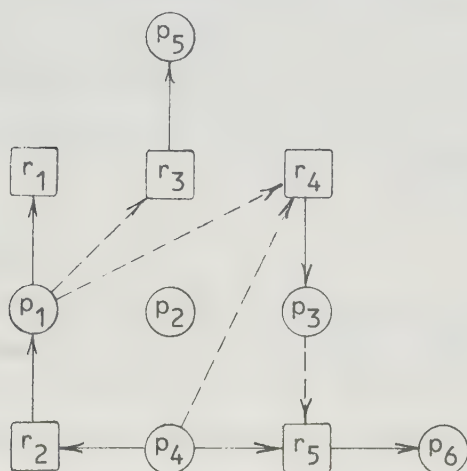
In the case of inserting a new node into an empty place p_q (row p_q is empty) we first compute column $(V_{p_q}[-] + \sum_{(r_v, p_q) \in E} P_G[-, r_v])$ and

row p_q . Observe that the new node obtains the name p_q corresponding to the empty place. Row p_q is given by $\sum_{(p_q, r_w) \in E} (P_G[p_{R_G[r_w]}, -])$

$+ (V_{r_w} []^T)$. Then, after computing P_{p_q} , the insertion of node p_q is expressed by $P_G + P_{p_q}$ and setting $R_G[r_v] = q$ for the nodes r_v from which there are edges to p_q .

Example 9:

A) Consider the resource allocation graph G given in Example 4 and its representation P_G and R_G given in Example 5 and the path matrix P_{p_2} given in Example 7. The deletion of node p_2 , expressed by $P_{G'} = P_G - P_{p_2}$ and setting $R_{G'}[r_1]$ to zero, gives the graph G' represented by:



$r_1 r_2 r_3 r_4 r_5$

p_i

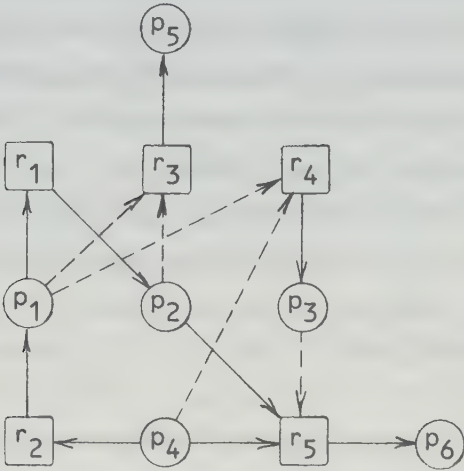
$$P_{G'} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}$$

$$R_{G'} = \begin{pmatrix} 0 \\ 1 \\ 5 \\ 3 \\ 6 \end{pmatrix} \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix}$$

B) Consider the resource allocation graph G' and its representation in part A). The insertion of a new node p_2 , represented by the edges (r_1, p_2) , (p_2, r_3) and (p_2, r_5) , is expressed by $P_{G''} = P_{G'} + P_{p_2}$ and setting $R_{G''}[r_1]$ to 2:

$$P_{p_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \otimes (0 \ 0 \ 1 \ 0 \ 1) = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and we obtain



$r_1 r_2 r_3 r_4 r_5$

p_i

$$P_{G''} = \begin{bmatrix} 1 & 0 & 2 & 1 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 2 & 2 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}$$

$$R_{G''} = \begin{bmatrix} 2 \\ 1 \\ 5 \\ 3 \\ 6 \end{bmatrix} \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix}$$

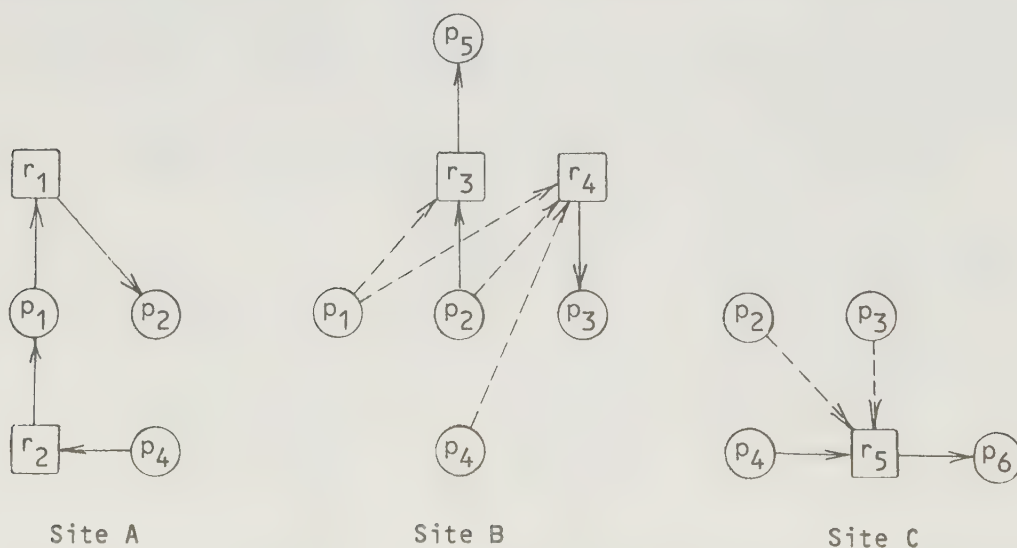
□

The deletion and insertion of a node r_q can be done in the same manner.

4. Application of the Technique to Distributed Systems

In a distributed system, with the processes acting at many sites, the resource allocation graph also becomes distributed. At each site, the subgraph is determined by the resource-nodes corresponding to the resources residing at the site. The subgraph then consists of these resource-nodes, process-nodes to/from which there are edges from/to the resource nodes, and of the edges between the nodes. The same process may act at different sites and we call the occurrence of a process at another site an agent of the process residing at the particular site. One possible splitting of the resource allocation graph in Example 4 is shown in Example 10. Here, process p_2 uses resource r_1 at site A and an agent is waiting for resource r_3 at site B. In the future, an agent of process p_2 will need resource r_4 at site B and another agent resource r_5 at site C. A similar scenario can be devised to account for each process of the resource allocation graph.

Example 10: A possible splitting of the resource allocation graph in Example 4.



4.1 The Centralized Approach

In the centralized approach a global resource allocation graph is maintained by a designated process at some site. Every resource request must be chaneled through this process. No subgraphs are maintained at the other sites. This case is similar to the case of multiprocessor systems treated in [5].

To express the running time of the algorithms we count the number of r- and p-nodes which can be reached from a node and the number of r- and p-nodes from which a node can be reached. Let

$\text{reach_to_r}(v) = |\{r_i; \text{there is a path of length } > 0 \text{ from } v \text{ to } r_i\}|$,
 $\text{reach_to_p}(v) = |\{p_i; \text{there is a path of length } > 0 \text{ from } v \text{ to } p_i\}|$,
 $\text{reach_from_r}(v) = |\{r_i; \text{there is a path of length } > 0 \text{ to } v \text{ from } r_i\}|$ and
 $\text{reach_from_p}(v) = |\{p_i; \text{there is a path of length } > 0 \text{ to } v \text{ from } p_i\}|$.

The running time of the algorithm for the attempted but unsuccessful insertion of an edge (r_v, p_w) is $O(1)$, while for the successful insertion or deletion of the edge it is

$O(\max(n, m, \text{reach_from_p}(r_v) * \text{reach_to_r}(p_w)))$.

In resource allocation systems rejected requests, that is, unsuccessful insertion of edges, are the most frequent operations [3]. The $O(1)$ cost of the unsuccessful insertions can amortize the cost of the other two types of operation and linear or even constant amortized time for one operation is realistic by maintaining the resource allocation graph. However, the sending and receiving of the messages from/to the site containing the process (which maintains the resource allocation graph) may become a bottleneck.

4.2 The Fully Distributed Approach

4.2.1 Introduction

The centralized deadlock avoidance algorithm requires all information to reside at one site. An alternative is to distribute the information among several sites. The hierarchical scheme given by Menasce and Muntz [26] would be easy to apply to our model. In their scheme the information is distributed to controllers structured in a tree, where the leaves of the tree correspond to the sites in the system. At rejected resource requests (unsuccessful allocations) the absence of cycles may be checked at $O(1)$ cost at every level in the tree in our model. Thus, the cost of one rejected request is $O(\log s)$, where s is the number of sites in the system. The number of messages sent by the operation is also $O(\log s)$. We will not say more about this method, but instead describe a fully distributed scheme where every site has the same controlling task.

Our representation of the resource allocation graph makes it easy to maintain the split graph. For that, we need the following information to perform the three operations, unsuccessful allocation, successful allocation, and release of resources to/from processes at each site:

- column r_i of the global resource allocation graph P_G and element r_i of the resource allocation vector R_G for each resource R_i residing at the site, and
- row p_j of the global resource allocation graph P_G for each process p_j residing at the site.

According to Theorem 5, we can then compute $P_{(r_v, p_w)}$ when inserting or

deleting an edge (r_v, p_w) . The insertion of an edge (r_v, p_w) can be expressed by $P_G + P_{(r_v, p_w)}$ and by setting $R_G[r_v]$ to w . After compu-

ting $P_{(r_v, p_w)}$ at the site which contains r_v , the site distributes

the addition of the path matrix $P_{(r_v, p_w)}$ among the sites. In the

same way, the deletion of an edge (r_v, p_w) can be expressed by $P_G - P_{(r_v, p_w)}$, by setting $R_G[r_v]$ to zero, and distributing the subtraction among the sites.

When a site is distributing the result of an operation it needs a list containing the indices of the rows and columns which will be sent to each site. In the following we assume that each site has this information (and this is "realistic" assumption).

In Sections 2 and 3 we gave a general description of our technique. In the following we describe the three operations in a distributed system where we do not differentiate between claim and request edges (not in the figures either). Our model makes it possible to compute the whole operation on an edge at the site which contains the edge. We assume that when attempting to allocate a resource r_v to a process p_w a request edge (p_w, r_v) always exists. Then, the allocation operation means converting the request edge (p_w, r_v) to an assignment edge (r_v, p_w) , which is later reconverted to (p_w, r_v) when r_v is released by p_w .

When attempting to allocate a resource r_v , the site which contains resource r_v in most cases has the necessary information for detecting whether the allocation makes the global graph cyclic. When a cycle is "indicated", which happens most frequently, the cost of the operation (unsuccessful allocation) is $O(1)$. When no cycle is "indicated" the allocation of the resource is possible, during which the mutual exclusion condition must hold, that is, no operation which changes the global graph must occur at the same time. The critical section consists of checking whether the allocation still is possible and if so, computing $P_{(p_w, r_v)}$ and $P_{(r_v, p_w)}$, and sending the appropriate rows

and columns of $(-P_{(p_w, r_v)} + P_{(r_v, p_w)})$ (corresponding to the conversion of the edge) to each site, adding these rows and columns to the part of the global path matrix at each site. In the following we will use the expressions $(-P_{(p_w, r_v)} + P_{(r_v, p_w)})$ or $(-P_{(r_v, p_w)} + P_{(p_w, r_v)})$

where the first term is computed first and already subtracted from the global path matrix when the computation of the second term starts.

4.2.2 The Modified Mutual Exclusion Algorithm

Algorithms for mutual exclusion in a distributed system are presented in [21,29,34]. To our purpose a modification of the algorithm proposed by Ricart and Agrawala in [29] can be used. In their algorithm, two types of messages can be sent: REQUEST messages to ask for permission to enter the critical section, and REPLY messages to give permission to enter the critical section. In our application, a site, when entering its critical section, may need the result of operations made in the critical sections at other sites. For this purpose, we introduce a third type of messages: ACKNOWLEDGEMENT messages, in which the result of the operations made in a critical section can be sent. In the following description we do not give all details from the algorithm in [29]. For a more formal description of the modified algorithm see Algorithm 1 in Section 4.2.3.

The Modified Mutual Exclusion Algorithm:

A site enters the first phase of its critical section after all other sites have been notified of the request and have sent a reply granting their permission. It then waits for the acknowledgements which has not yet arrived. An ACKNOWLEDGEMENT message can contain appropriate rows and columns to be added to the local part of the global path matrix at the site. The addition sent in the message is a part of the critical section of the site which sent the acknowledgement. An ACKNOWLEDGEMENT can also be "empty", that is, not contain any addition to be performed. The second phase of the critical section starts when all acknowledgement messages are received and the operations sent are performed.

In the second phase of the critical section, the site checks whether the allocation of the resource is still possible. If so, the site computes $(-P_{(p_w, r_v)} + P_{(r_v, p_w)})$ and in an ACKNOWLEDGEMENT message

sends the appropriate rows and columns to each site for addition. The sites receiving the ACKNOWLEDGEMENT message immediately perform the addition of the sent rows and columns. If the allocation is no longer possible (because the allocation of another resource has caused a path from p_w to r_v) the site sends an empty ACKNOWLEDGEMENT

to all sites. An ACKNOWLEDGEMENT message is sent together with a REPLY message to the sites which are waiting for the REPLY message.

A site making an attempt to invoke mutual exclusion sends a REQUEST message to all other sites. Upon receipt of the REQUEST message, the other sites either sends a REPLY immediately or defers a response until after it leaves its own critical section.

The algorithm is based on the fact that a site receiving a REQUEST message can immediately determine whether the requesting site or itself should be allowed to enter its critical section first. The site sending the REQUEST message is not told the result of the comparison, but a REPLY message is returned immediately if the sender of the REQUEST message has priority; otherwise, the REPLY is delayed. The priority order decision is made by comparing a sequence number present in each REQUEST message. If the sequence numbers are equal, the site numbers are compared to determine which will enter first. $\square$

Note that the first phase of the critical section at a site is the the end of the critical sections at the other sites. We need this first phase because messages are not guaranteed to be delivered in the same order in which they are sent, which means that a reply message sent by a site A can arrive to site B before a previously sent acknowledgement message from site A. The second phase of the critical section is the "real" critical section. Furthermore, the addition of the rows and columns sent in an ACKNOWLEDGEMENT message should be immediately performed since the checking whether the allocation of a resource is possible needs the latest status of the global resource allocation graph.

Ricart and Agrawala [29] have shown that their algorithm exhibits the following desirable behaviour:

- a) Mutual-exclusion is obtained.
- b) Freedom from deadlock is ensured.
- c) Freedom from starvation is ensured.
- d) The number of messages required is $2 \cdot (s - 1)$, where s is the number of sites in the system per critical section invocation.

Our modification means that a site exiting its critical section sends an ACKNOWLEDGEMENT message to each site, which can be sent together with a REPLY message. Furthermore, the allocation of a resource can fail even though mutual-exclusion is achieved.

To avoid starvation when the allocation of a resource has failed, a later attempted allocation of the same resource to the same process (when no cycle is indicated) obtains the earlier sequence number, which will be the lowest (meaning that the site has priority over the other sites) after some period of time. However, it may still happen that previously granted permissions to allocate resources at other sites cause a path which prevents the allocation of the resource to the starving process. This problem can be solved by deferring all requests for allocation after an appropriately large difference between the sequence number of the allocation and the other requests. After some period of time some process will then be released making the allocation of the starving process possible.

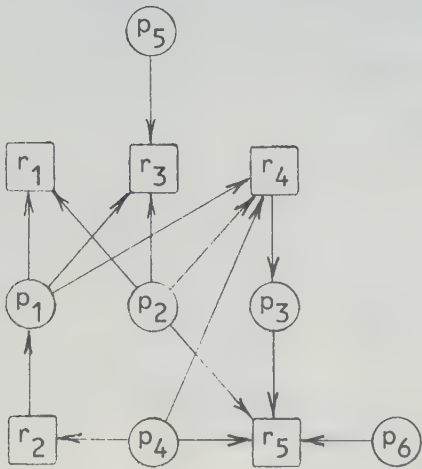
The number of messages required for one critical section is greater in our modification since a site exiting its critical section sends an ACKNOWLEDGEMENT message to each site. However, the acknowledgements are sent together with replies when possible, and consequently the number of messages required varies between $2*(s - 1)$ and $3*(s - 1)$. It is clear that mutual-exclusion is obtained since a site enters its "real" critical section only after it has received all acknowledgements and performed the operations. This means that the other sites have left their critical sections and the global graph cannot be changed by another site in the meantime.

It should also be clear that our modification does not affect the deadlock freedom of the original algorithm.

4.2.3 Allocation and Release of Resources to/from Processes

We first give an example and then the algorithm for an attempted allocation of a resource r_v to a process p_w .

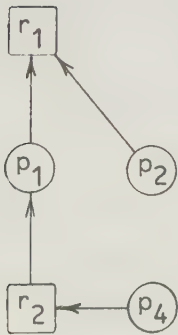
Example 11: Consider the following global resource allocation graph G and its representation:



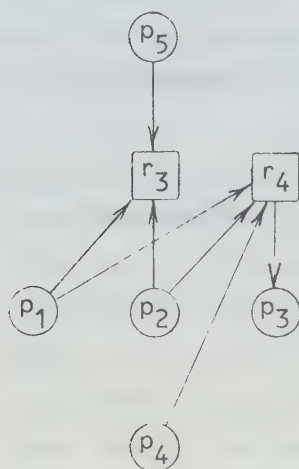
$$P_G = \begin{matrix} & r_1 & r_2 & r_3 & r_4 & r_5 \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 3 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix} \end{matrix} \quad R_G = \begin{matrix} & p_i \\ \begin{pmatrix} 0 \\ 1 \\ 0 \\ 3 \\ 0 \end{pmatrix} & \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{matrix} \end{matrix}$$

The global graph is split in three parts located at sites A, B and C. At each site only the necessary rows and columns are represented:

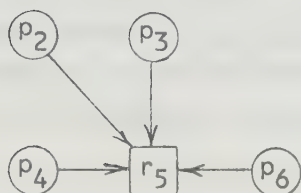
Site A:



$$P_G = \begin{matrix} & r_1 & r_2 \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 2 \\ 0 & 0 & & & \\ 1 & 1 & 1 & 2 & 3 \\ 0 & 0 & & & \\ 0 & 0 & & & \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ \\ p_4 \\ \\ \end{matrix} \end{matrix} \quad R_G = \begin{matrix} & p_i \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \begin{matrix} r_1 \\ r_2 \end{matrix} \end{matrix}$$

Site B:

$$P_G = \begin{matrix} & r_3 & r_4 & \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 3 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & & & \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{matrix} \end{matrix} \quad R_G = \begin{matrix} & p_i & \\ \begin{pmatrix} 0 \\ 3 \end{pmatrix} & \begin{matrix} r_3 \\ r_4 \end{matrix} \end{matrix}$$

Site C:

$$P_G = \begin{matrix} & r_5 & \\ \begin{pmatrix} & & 1 \\ 1 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 3 \\ & & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} & \begin{matrix} p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix} \end{matrix} \quad R_G = \begin{matrix} & p_i & \\ (0) & r_5 \end{matrix}$$

Consider the following scenario:

1. Site C attempts to allocate r_5 to p_4 but fails since $P_G[p_4, r_5] > 1$.
2. Site B requests to release r_4 from p_3 (which means reconverting the edge (r_4, p_3) to (p_3, r_4)). It sends a REQUEST to Sites A and C and then receives a REPLY from them. Since Site B has received all ACKNOWLEDGEMENT messages corresponding to previously sent REPLY messages, it can enter the critical section for releasing r_4 .
3. Site B requests to allocate resource r_3 to process p_5 which is possible since $P_G[p_5, r_3] \neq 1$. It sends a REQUEST to Sites A and C and receives a REPLY from them. But, it cannot enter this critical

section since the execution of its previous critical section is not completed.

4. Site B has finished the computation of $(-P_{(r_4, p_3)} + P_{(p_3, r_4)})$:

$$P_{(r_4, p_3)} = P_G[-, r_4] \otimes P_G[p_3, -] = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 2 \\ 0 \\ 0 \end{pmatrix} \otimes (0 \ 0 \ 0 \ 0 \ 1) = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$P_G := P_G - P_{(r_4, p_3)} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & & & \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{matrix} \quad R_G = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{matrix} r_3 \\ r_4 \end{matrix}$$

In a situation, when a resource r_w is not allocated to any process the vector $P_G[p_{R_G[r_w]}, -]$ in the expression, which gives the path

matrix $P_{(p_v, r_w)}$, obtains the index p_0 . This vector is considered as a

vector consisting of zero elements.

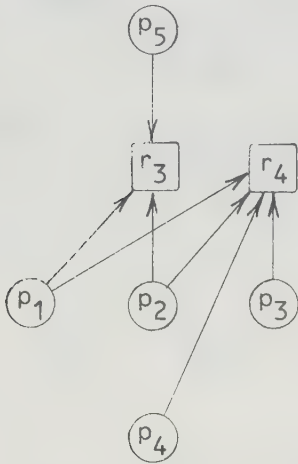
$$P_{(p_3, r_4)} = V_{p_3}[-] \otimes (P_G[p_0, -] + (V_{r_4}[-])^T) =$$

$$= \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \otimes (0 \ 0 \ 0 \ 1 \ 0) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$(-P_{(r_4, p_3)} + P_{(p_3, r_4)}) = \begin{pmatrix} 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

It then sends the appropriate rows and columns in an ACKNOWLEDGEMENT message to Sites A and C, and updates its part of the global path matrix and resource allocation vector:

Site B:



$$P_G = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & & & \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{matrix}$$

$$R_G = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{matrix} r_3 \\ r_4 \end{matrix}$$

5. Site B checks whether the allocation of r_3 to p_5 is still possible. Since it is possible it enters the critical section for allocating r_3 . Sites A and C receive the ACKNOWLEDGEMENT message from Site B and update their part of the global path matrix P_G .

Site A:

$$P_G = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & & & \\ 1 & 1 & 1 & 2 & 1 \\ 0 & 0 & & & \\ 0 & 0 & & & \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ \\ p_4 \end{matrix}$$

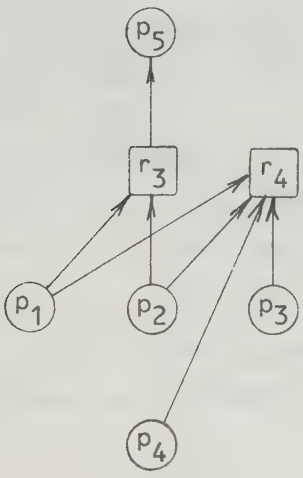
Site C:

$$P_G = \begin{pmatrix} & & & & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ & & & & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} \\ p_2 \\ p_3 \\ p_4 \\ \\ p_6 \end{matrix}$$

6. Site A attempts to allocate r_1 to p_2 and Site C r_5 to p_4 and both sends a REQUEST to the other sites. Suppose C has the priority and defers to send a REPLY to A while itself receives a REPLY from A. Neither A or C receives any REPLY from B since it is in its critical section.
7. Site B has finished the computation of $(-P_{(p_5, r_3)} + P_{(r_3, p_5)})$.

It sends the appropriate rows and columns in an ACKNOWLEDGEMENT message together with a REPLY message to Sites A and C. It updates its part of the global path matrix and resource allocation vector:

Site B:



$$P_G = \begin{matrix} & r_3 & r_4 & & p_i \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & & \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{matrix} & R_G = \begin{pmatrix} 5 \\ 0 \end{pmatrix} \begin{matrix} r_3 \\ r_4 \end{matrix} \end{matrix}$$

Site A and C receive the ACKNOWLEDGEMENT and REPLY messages and update their part of the global path matrix P_G :

Site A:

Site C:

$$P_G = \begin{matrix} & r_1 & r_2 & & p_i \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & & & \\ 1 & 1 & 1 & 2 & 1 \\ 0 & 0 & & & \\ 0 & 0 & & & \end{pmatrix} & \begin{matrix} p_1 \\ p_2 \\ \\ p_4 \end{matrix} \end{matrix}$$

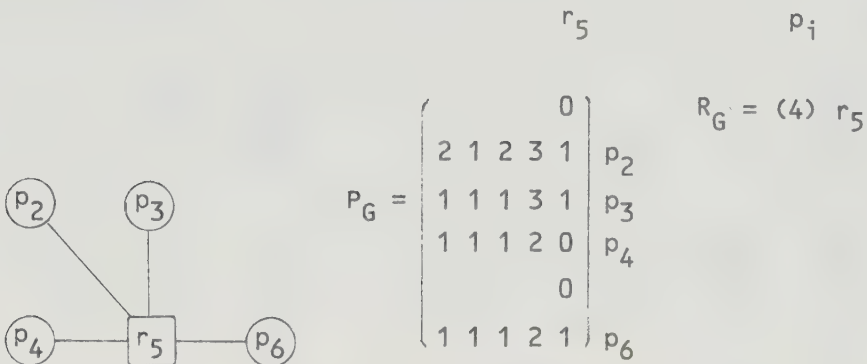
$$P_G = \begin{matrix} & r_5 & & & p_i \\ \begin{pmatrix} & & & & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ & & & & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} & \begin{matrix} \\ p_2 \\ p_3 \\ p_4 \\ \\ p_6 \end{matrix} \end{matrix}$$

8. Site C has the permission from both sites to enter its critical section. It first checks whether the allocation is still possible. Since it is possible it enters the critical section for allocating r_5 to p_4 .

9. Site C has finished the computation of $(-P_{(p_4, r_5)} + P_{(r_5, p_4)})$.

It sends the appropriate rows and columns in an ACKNOWLEDGEMENT message to Sites A and C, to Site A together with a REPLY. It updates its part of the global path matrix and resource allocation vector:

Site C:



Sites A and B receive the ACKNOWLEDGEMENT message (Site A even the REPLY message) and update their part of the global path matrix P_G :

Site A:

Site B:

$$P_G = \begin{pmatrix} & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}$$

$$P_G = \begin{pmatrix} & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix} \begin{matrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{matrix}$$

10. Site A now has permission from both B and C to allocate, and has also received the ACKNOWLEDGEMENT messages for its replies. It checks whether the allocation of r_1 to p_2 is still possible.


```

(18)          Send the appropriate rows and columns of
              ( $- P_{(p_w, r_v)} + P_{(r_v, p_w)}$ ) to be added to the

              other sites in an ACKNOWLEDGEMENT message,
              eventually together with a deferred REPLY
              message

(19)          end

              (* exit the critical section          *)

(20)          end

```

Theorem 6: Assume that a site contains element r_i of the resource allocation vector R_G and column r_i and row p_j of the global path matrix P_G for each resource r_i and process p_j residing at the site. Assume further that by the computation of ($- P_{(p_w, r_v)} + P_{(r_v, p_w)}$)

at row (16) in Algorithm 1, $P_{(p_w, r_v)}$ is computed first, then, the

appropriate rows and columns are subtracted from the local part of P_G , and finally, the new values of the local part of P_G are used to compute $P_{(r_v, p_w)}$. Furthermore, assume that the site after receiv-

ing an ACKNOWLEDGEMENT message immediately performs the operation sent. When the site attempts to allocate a resource r_v to a process p_w Algorithm 1 then guarantees that the system is kept in a safe state.

Proof: Theorem 5 gives that the site has the necessary information to compute the conversion of the edge (p_w, r_v) to the edge (r_v, p_w) in the global resource allocation graph. The global graph can only be changed when a site is in its critical section for converting an edge. In that case the mutual-exclusion is obtained since the site enters its "real" critical section only after it has received all acknowledgements, which means, that the other sites have left their critical sections and the global graph cannot be changed by another site. According to Theorem 4, it can then correctly be checked at lines (2) and (9) whether the allocation of resource r_v to process p_w keeps the system in a safe state. $\square$

We will use the definitions of concurrence given in [28] in the next theorem. We consider all events executed at a single site to be totally ordered. Also, by the law of causality, a message can be received only after it has been sent. Therefore, we can define the happened before relation (denoted by $\rightarrow$) on a set of events as follows (assuming that sending and receiving a message constitutes an event):

1. If X and Y are events at the same site, and X was executed before Y , then $X \rightarrow Y$.
2. If X is the event of sending a message at one site and Y is the event of receiving that message at another site, then $X \rightarrow Y$.
3. If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$.

Since an event cannot happen before itself, the $\rightarrow$ relation is an irreflexive partial ordering. If two events, X and Y , are not related by the $\rightarrow$ relation (that is, X did not happen before Y , and Y did not happen before X), then we say that these two events were executed concurrently. In this case, neither event can causally affect the other. If, however, $X \rightarrow Y$, then it is possible for event X to causally affect Y .

In the following we consider the administration cost of the mutual-exclusion algorithm as a cost included in the cost of sending the messages.

To save time when computing $(-P_{(p_w, r_v)} + P_{(r_v, p_w)})$ in Algorithm 1

we only compute and then subtract and add the non-zero elements of $P_{(p_w, r_v)}$ and $P_{(r_v, p_w)}$ respectively to the local part of the path matrix P_G at each site.

Theorem 7: Suppose that only the non-zero elements of $(-P_{(p_w, r_v)} + P_{(r_v, p_w)})$ are computed at line (16), and only the non-zero elements of the appropriate rows and columns of $(-P_{(p_w, r_v)} + P_{(r_v, p_w)})$, together with their indices are sent at line (18) in Algorithm 1. Suppose further that the appropriate rows and columns are represen-

- ted as arrays at each site and that the number of sites is s . Then,
- A) The total cost of an unsuccessful allocation is $O(1)$ in computational cost, and the number of sent messages is zero when no successful allocation is executed concurrently at another site causing a path from p_w to r_v in the global graph.
 - B) The total cost of an unsuccessful allocation is $O(1)$ in computational cost and the number of sent messages is between $2*(s - 1)$ and $3*(s - 1)$ when some successful allocation is executed concurrently at another site causing a path from p_w to r_v in the global graph.
 - C) The total cost of an successful allocation is $O(s * \min(\max(n * (\text{indeg}(p_w) + 1), m, \text{reach_from_p}(p_w) * \text{reach_to_r}(r_v)), \max(n, m, \text{reach_from_p}(r_v) * \text{reach_to_r}(p_w))))$ in computational cost with the number of sent messages between $2*(s - 1)$ and $3*(s - 1)$.

Proof:

- A) It follows immediately from Algorithm 1 and the definition of concurrence.
- B) It follows immediately from Algorithm 1, the description of the Modified Mutual Exclusion Algorithm, the argument following it, and from the definition of concurrence.
- C) It is shown in [5] that the cost of the deletion of an edge (p_w, r_v) is $O(\max(n * (\text{indeg}(p_w) + 1), m, \text{reach_from_p}(p_w) * \text{reach_to_r}(r_v)))$ and the cost of the insertion of an edge (r_v, p_w) is $O(\max(n, m, \text{reach_from_p}(r_v) * \text{reach_to_r}(p_w)))$. The rest of the proof follows immediately from Algorithm 1, the description of the Modified Mutual Exclusion Algorithm and the argument after. $\square$

The release of a resource r_v from a process p_w is always possible and the mutual-exclusion condition must hold during the operation. This operation can be done in the same manner as the successful allocation and at the same total_cost.

Unsuccessful allocations are the most frequent operations in a resource allocation system. In our distributed system, the $O(1)$ computational cost of an unsuccessful allocation can amortize the computational cost of the other two types of operation. When for each

release and successful allocation operation there are $O(\max(n*m))$ unsuccessful allocations, the amortized cost of one operation is linear per site, and when there are $O(n*m)$ unsuccessful allocations, the amortized cost of one operation is constant per site. The number of sent messages is zero at an unsuccessful allocation when no successful allocation is executed concurrently at another site causing a path from p_w to r_v in the global graph. The occurrence of this zero number messages sent by an unsuccessful operation depends on the speed of the message traffic. With sufficiently high speed, this zero number of sent messages can amortize the number of messages sent by the other operations and an amortized constant number of sent messages is realistic per operation.

4.2.4 Allocation and Release of Processes and Resources to/from the System

When a new process p_q enters the system, it may be allocated to the local part of the global resource allocation graph at several sites (according to the locality of its claimed resources). The process may enter the system (by performing the allocation) at any site. For "simplicity", we assume that the allocation of the process is only performed at one site and that no resource is being allocated to the process at the same time. The allocation of the process to the system is then expressed by adding P_{p_q} (corresponding to the paths starting in p_q) to

the global path matrix P_G . Since no paths are ending in p_q , corresponding to the assumption above, we obtain the path matrix

$P_{p_q} = V_{p_q}[_] \otimes P_G[p_q, _]$ where only row p_q contains non-zero elements.

Row p_q (i.e., $P_G[p_q, _]$) is given by the sum of the rows $P_G[p_{R_G}[r_w], _]$ and $(V_{r_w}[_])^T$, corresponding to nodes r_w ,

$w = 1, 2, \dots, m$, which have edges from p_q .

The "real" critical section for allocating p_q to the system consists of the computation of row $P_G[p_q, _]$ and sending it to appropriate sites. The rows $P_G[p_{R_G}[r_w], _]$ are not always present at the site

performing the allocation. Therefore, the previously given mutual-exclusion algorithm must be extended. The easiest extension is that each site contains the whole path matrix P_G , and when converting an edge, the non-zero elements of the whole path matrix, corresponding to the conversion, are sent to each site. This extension implies increased computational cost, but the cost is asymptotically the same as given in Theorem 7. An alternative is to request the wanted rows and ensure the allocating site to receive these rows correctly updated. In the following, we give a description of this alternative, where the row $P_G[p_{R_G[r_w]}, -]$ is requested from the site which contains

resource r_w . Furthermore, we assume that conditions given in the Modified Mutual Exclusion Algorithm are valid here, for example, a site defers to send replies when it is in its critical section (see Section 4.2.2).

Extension of the Modified Mutual Exclusion Algorithm:

A site performing the allocation of a process p_q to the system sends special requests, denoted by $P_REQUEST$ to each site, eventually containing indices for wanted rows. A $P_REQUEST$ has the priority over a $REQUEST$.

After receiving a $P_REQUEST$ a site defers to send $REPLY$ messages. To ensure the wanted rows are sent correctly updated, the site, which receives a $P_REQUEST$ with indices of wanted rows, has to wait for all $ACKNOWLEDGEMENT$ messages before sending a P_REPLY message containing the wanted rows. After the P_REPLY message is sent (with or without wanted rows), the site continues to defer sending $REPLY$ messages. However, after receiving another $P_REQUEST$, another P_REPLY can immediately be sent since the allocation of process p_q does not have any effect on other rows than row p_q based on the assumption that no resources are allocated to p_q and the allocation of p_q is only performed at one site.

When the site performing the allocation of p_q has received all P_REPLY messages, it computes $P_G[p_q, -]$ and sends a $P_ACKNOWLEDGEMENT$ message to each site, together with the non-zero elements of $P_G[p_q, -]$ to appropriate sites.

Then, finally after receiving all $P_ACKNOWLEDGEMENT$ messages a site begins to send $REPLY$ messages as "usual". $\square$

Note that a site has to defer sending REPLY messages until all P_ACKNOWLEDGEMENT messages have arrived to ensure that no other site alters the rows which were (eventually) requested in the P_REQUEST messages. We need this rule because the time it takes for a site to receive a message from another site can vary. Otherwise, it could happen, for example, that a Site B sends the requested rows in a P_REPLY to a Site A, and then a REPLY to a Site C which arrives to Site C before the P_REQUEST from Site A. Then, Site C can alter those rows at Site B which were sent by B to Site A in the P_REPLY message.

The critical section for allocation of a process p_q to the system consists of receiving the requested rows, computing row $P_G[p_q, -]$ and sending it to appropriate sites. Since the allocation of a process p_q does not have any effect on other rows than $P_G[p_q, -]$, the mutual-exclusion condition must only hold between the critical section for allocating a process to the system and the critical section for allocating a resource to a process. It is easy to see that P_{p_q} is

correctly computed, that is, the site which is allocating p_q receives the correctly updated rows $P_G[p_{R_G[r_w]}, -]$, since

- the sending sites first receive all acknowledgements to their previously sent replies, and then
- send the P_REPLY message together with the wanted rows, and
- defer sending REPLY messages until the P_ACKNOWLEDGEMENT message is received.

Furthermore, the desired mutual-exclusion condition between allocation of process p_q to the system and allocation of resources to processes holds since

- the site which is allocating p_q receives all P_REPLY and ACKNOWLEDGEMENT messages before entering its "real" critical section, implying that no other site can be in its critical section for allocating a resource to a process, and
- the other sites can enter their "real" critical section for allocating a resource to a process first after receiving a P_ACKNOWLEDGEMENT message from the site, corresponding to the end of the critical section of the site.

It is clear that the extension cannot cause starvation since the processes recently allocated to the system have to request resour-

ces as before. Neither can the extension cause deadlock since the allocation of processes to the system has priority over allocation of resources to processes, and since the allocation of processes can be done concurrently.

The number_of_messages_required is the same as for the modified algorithm, that is, it varies between $2*(s - 1)$ and $3*(s - 1)$.

Note that this extension implies that no REPLY messages are sent in the system while some site is in the critical section for allocating a process to the system.

The total cost of allocating a process p_q to the system is $O(s*m*outdeg(p_q))$ in computational cost according to [5] and to the fact that row p_q may be sent to each site. The number_of_sent_messages is between $2*(s - 1)$ and $3*(s - 1)$.

When releasing a process p_q from the system we assume that no resources are allocated to the process. Then, $P_{p_q} = V_{p_q}[_] \otimes P_G[p_q, _]$

and the releasing of p_q from the system is expressed by setting the elements of $P_G[p_q, _]$ to zero. This can be done in the same manner as the allocation of a process. We obtain an $O(s*m)$ computational_cost with the number_of_sent_messages between $2*(s - 1)$ and $3*(s - 1)$.

Note that the rule that a P_REQUEST has priority over a REQUEST can neither in this case cause starvation since there are only a finite number of processes in the system to be released.

When allocating a resource r_w to the system we assume that the resource is not immediately allocated to any process and that no process is requesting the resource. The mutual-exclusion condition does not have to hold since the allocation does not change the value of any element of the path matrix P_G . The allocation then means sending a message about the existence of the resource to each site, at total cost $O(s)$ in computational_cost and s , the number_of_sent_messages.

We can make the same assumption and perform the operation in a similar way when the release of a resource is anticipated.

4.2.5 Unexpected Release of Processes and Resources

So far, we considered the allocation and release of processes and resources to/from the system as anticipated. However, it may happen that some resource fails, leading to that the resource and eventually several processes must be unexpectedly released from the system. Our representation of the resource allocation graph also gives a clear picture in this case. Now, we cannot make the assumption, when releasing a resource, that no process is requesting the resource and that the resource is not allocated to any process. Furthermore, we cannot assume that no resource is allocated to a process to be released. It means that the subtraction of the path matrices

P_{r_w} and P_{p_q} may affect several rows of the global path matrix P_G ,

implying that the mutual-exclusion condition between this type of operations must hold.

To avoid deadlock when several sites are in this situation, the priority between P_REQUESTs (or R_REQUESTs in the case of resources) of this type can be decided by the sequence number mentioned in the description of the Modified Mutual Exclusion Algorithm.

We obtain a total cost $O(s \cdot \max(n, m, \text{reach_from_p}(r_w) \cdot \text{reach_to_r}(r_w)))$ for the release of a resource r_w , and

$O(s \cdot \max(n \cdot (\text{indeg}(p_q) + 1), m, \text{reach_from_p}(p_q) \cdot \text{reach_to_r}(p_q)))$ for the release of a process p_q in computational cost according to [5].

The number of sent messages is between $2 \cdot (s - 1)$ and $3 \cdot (s - 1)$.

5. Conclusion

Our dynamic technique gives the a very efficient solution of the deadlock avoidance problem in a fully distributed environment.

Since the global resource allocation graph, represented by a path matrix, is easy to partition, each site can contain sufficient information to efficiently detect cycles. Furthermore, the representation gives full control over such unexpected situations as when some failing resource leads to that the resource and eventually several processes must be released from the system.

A comparison between our technique and topological sorting, which is the previously proposed technique for cycle detection, is presented in [4]. The comparison is made for acyclic digraphs when the number of edges e is $\theta(n^2)$ (where n is the number of nodes in the graph) and is relevant to resource allocation graphs. The comparison shows that our technique is superior in the case of operations on edges; it is better by an additive term in deletion and successful insertion while in unsuccessful insertion it is much better. It gives an $O(1)$ running time while topological sorting gives $\theta(n + e)$.

In the case of operations on nodes, our technique is better when for each deletion and successful insertion there are more than $0.75 \cdot n$ unsuccessful insertions, which is a realistic assumption for resource allocation systems.

A simulation presented in [3] shows that rejected requests are the most frequent operations in a resource allocation system. It means, with our resource allocation algorithms operating on edges, that unsuccessful insertions are most frequent of the three operations. The $O(1)$ cost of one unsuccessful insertion operation can amortize the cost of the other two types of operation. When for each release and successful allocation there are $O(\max(n, m))$ unsuccessful allocations, the amortized cost of one operation is linear per site, and when there are $O(n \cdot m)$ unsuccessful allocations, the amortized cost of one operation is constant per site. This latter situation may very well happen in overloaded resource allocation systems. The number of sent messages is zero at an unsuccessful allocation of a resource r_v to a process p_w , when no successful allocation is executed concurrently at another site causing a path from p_w to r_v in the global graph. The occurrence of the zero number messages sent by an unsuccessful al-

location depends on the speed of the message traffic. With sufficiently high speed, this zero number of sent messages can amortize the number of sent messages sent by the other operations and an amortized constant number of messages is realistic per operation.

Future systems will be oriented more toward asynchronous parallel operation than toward the serial systems of the past. Multiprocessing will be common and parallel computation will be prevalent. There will, quite simply, be more operations going on concurrently. Resource allocation will tend to be dynamic and processes will be able to acquire and release resources freely as needed. With the increasing tendency of operating systems designers to view data as a resource the number of resources operating systems must manage will increase dramatically. In the dynamic systems of the future, rejection of requests will occur very frequently. Our dynamic technique with its $O(1)$ running time and zero number of sent messages in most cases for a rejection operation fits future systems very well.

Acknowledgement

The author is grateful to Dr Rolf Karlsson for his useful suggestions.

References

1. A. V. Aho & J. E. Hopcroft & J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1975.
2. A. V. Aho & J. E. Hopcroft & J. D. Ullman, "Data Structures and Algorithms", Addison-Wesley, 1983.
3. F. Belik, "Deadlock Avoidance with a Modified Banker's Algorithm", LUNFD6/(NFCS-3008), Dept. of Computer Science, Lund University, Sweden, 1985.
4. F. Belik, "Dynamic Operation on Acyclic Digraphs", LUNFD6/(NFCS-3010), Dept. of Computer Science, Lund University, Sweden, 1985.
5. F. Belik, "An Efficient Deadlock Avoidance Technique", Unpublished.
6. P. A. Bernstein & N. Goodman, "Concurrency Control in Distributed Database Systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.
7. P. A. Bernstein & J. B. Rothnie & N. Goodman & C. A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1 : A System for Distributed Databases", IEEE Trans. on Software Engineering, Vol. SE-4, No. 3, May 1978, pp. 154-168.
8. G. N. Buckley & A. Silberschatz, "Beyond Two-Phase Locking", Journal of the ACM, Vol. 32, No. 2, April 1985, pp. 314-326.
9. M. C. Chen & M. Rem, "Deadlock-Freedom in Resource Contentions", Acta Informatica 21, 1985, pp. 585-598.
10. E. G. Coffman & M. J. Elphick & A. Shoshani, "System Deadlocks", Computing Surveys, Vol. 3, No. 2, June 1971, pp. 67-78.
11. H. M. Deitel, "An Introduction to Operating Systems", Addison-Wesley, 1984.
12. E. W. Dijkstra, "Cooperating Sequential Processes", Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.
13. V. D. Gligor & S. H. Shattuck, "On Deadlock Detection in Distributed Systems", IEEE Trans. on Software Engineering, Vol. SE-6, No. 5, September 1980, pp. 435-440.
14. A. N. Haberman, "Prevention of System Deadlocks", Communications of the ACM, Vol. 12, No. 7, July 1969, pp. 373-385.

15. J. W. Havender, "Avoiding Deadlock in Multitasking Systems", IBM System Journal, Vol. 7, No. 2, 1968, pp. 74-84.
16. R. C. Holt, "Some Deadlock Properties of Computer Systems", Computing Surveys, Vol. 4, No. 3, September 1972, pp. 179-196.
17. Toshihide Ibaraki & Tsunehiko Kameda, "Deadlock-Free Systems for Bounded Number of Processes", IEEE Trans. on Computers, Vol. C-31, No. 3, March 1982, pp. 188-193.
18. Tiko Kameda, "Testing Deadlock-Freedom of Computer Systems", Journal of the ACM, Vol. 27, No. 2, April 1980, pp. 270-280.
19. D. E. Knuth, "The Art of Computer Programming", Volume 1, Addison-Wesley, 1975.
20. W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.
21. L. Lamport, "Time, Clocks, and the Ordering of Events in Distributed System", Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 558-565.
22. J. Y.-T. Leung & E. K. Lai, "On Minimum Cost Recovery from System Deadlock", IEEE Trans. on Computers, Vol. C-28, No. 9, September 1979, pp. 671-677.
23. D. B. Lomet, "Subsystems of Processes with Deadlock Avoidance", IEEE Trans. on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 297-304.
24. H. Madduri & R. Finkel, "Extension of the Banker's Algorithm for Resource Allocation in a Distributed Operating System", Inf. Process. Lett. 19, 1 (July 1984), pp. 1-8.
25. K. Mehlhorn, "Graph Algorithms and NP-Completeness", Springer Verlag, 1984.
26. D. A. Menasce & R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 195-202.
27. R. Obermark, "Distributed Deadlock Detection Algorithm", ACM Transactions on Database Systems, Vol. 7, No. 2, June 1982, pp. 187-208.
28. J. Peterson & A. Silberschatz, "Operating System Concepts", Addison-Wesley, 1985.
29. G. Ricart & A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", Communications of the ACM, Vol. 24, No. 1, January 1981, pp. 9-17.

30. D. J. Rosenkrantz & R. E. Searns & P. M. Lewis II, "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems, Vol. 3, No. 2, June 1978, pp.178-198.
31. D. J. Rypka & A. P. Lucido, "Deadlock Detection and Avoidance for Shared Logical Resources", IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, January 1985, pp. 67-80.
32. M. K. Sinha & N.Natarajan, "A Priority Based Distributed Deadlock Detection Algorithm", IEEE Trans. on Software Engineering, Vol. SE-11, No. 1, January 1985, pp. 67-80.
33. M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed I N G R E S", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 188-194.
34. I. Suzuki & T. Kasami, "A Distributed Mutual Exclusion Algorithm", ACM Transactions on Computer Systems, Vol. 3, No. 4, November 1985, pp. 344-349.
35. R. E. Tarjan, "Data Structures and Network Algorithms", SIAM Monograph, 1983.

CONCLUSIONS AND FURTHER RESEARCH

Our initial idea of constructing a dynamic set model, where movement of data can be expressed, came into use only in the first part of this thesis work. However, the attempts to use the model have led to a considerable improvement on a known algorithm and to another model which, in turn, produced new and very efficient algorithms in an important part of data processing.

While working on the thesis we got other practical and theoretical ideas. For example, we are now working on a project (supported by the National Swedish Board for Technical Development, STU) for implementing a data base application of the dynamic set model. We there construct a simple relational data base system with a query language which can run in two states; either as "usual" and or according to our model. The implementation is used to examine:

- how much the accessibility of the data base increases by using the model,
- how the accessibility varies with different type of relational algebra operations,
- how the time of locking the elements influences the increase of accessibility, and
- how much the increased accessibility costs in terms of administration of the model.

In future research, we intend to apply the dynamic graph model to the deadlock avoidance problem in systems with several instances of a resource type, and to precedence graphs for concurrent processes or transactions in data bases. Furthermore, when we were preparing the dynamic graph model we found a partial ordering of the nodes in an acyclic directed graph, which can be used to construct simpler algorithms for the transitive closure and all pairs shortest paths problems. A shortcoming of the dynamic set model is that it cannot express the entering and leaving of the elements of a set. The use of an implication relation,

called de Morgan implication, which can be characterized as a four-valued logic, makes it possible to express that elements leave or enter the set. We intend to develop a model with this logic and, among other things, study the possibilities of expressing communication between processes.

